Algoritmos y Estructuras de Datos

Cátedra de *Algoritmos y Estructuras de Datos*
http://www.cimec.org.ar/aed

Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral http://www.unl.edu.ar

Centro Internacional de Métodos Computacionales
en Ingeniería - CIMEC
INTEC, (CONICET-UNL), http://www.cimec.org.ar

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Autores

- **Mario Storti** *<mario.storti@gmail.com>*
- **Rodrigo Paz,** *<rodrigop@intec.unl.edu.ar>*
- **Lisandro Dalcín,** *<dalcinl@gmail.com>*
- **Jorge D'Elía,** *<jdelia@intec.unl.edu.ar>*
- **Martín Pucheta,** *<mpucheta@intec.unl.edu.ar>*

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Contents

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Dictado

- **Existe una página web con información y material del curso: `http://www.cimec.org.ar/aed`. Incluye material (apuntes y esta transparencia), y todo el código que se enseña durante el desarrollo del curso.**
- **Desde la página se puede suscribir a la lista de correo *`Noti-AED`*. Por la lista se manda información sobre el curso. Recomendamos que todos los alumnos deben estar *suscriptos a la lista.***
- **Es recomendable leer los apuntes *antes* de la clase correspondiente.**
- **Evaluación: *3 parciales*. Regulariza con $\geq$40% en los 3. Promueve si además el promedio es $\geq$70%.**
- **Habrá un *recuperatorio* que reemplaza la peor nota.**
- **El curso se dará en *lenguaje C++*, en especial el uso de la *librería STL* (hoy parte del C++ estándar). Es un objetivo del curso que los alumnos finalicen el curso con un buen nivel de programación en C++.**
- **De acuerdo con los lineamientos de le UNL *se fomentará el uso de Software Libre (ver `http://www.gnu.org`) durante todo el cursado.* En particular todos los ejemplos son probados con GCC/G++ en plataforma GNU/Linux.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos

En matemática y ciencias de la computación un *algoritmo* es un procedimiento (un conjunto de instrucciones bien definidas) para cumplir una tarea. A partir de un *estado inicial*, el algoritmo debe terminar en un *estado final* bien definido.

Informalmente, el concepto de algoritmo es similar al de una *receta de cocina*, si bien los algoritmos pueden ser mucho más complejos. Los algoritmos pueden tener pasos que se repiten (iteración) o requieren decisiones que involucran operaciones lógicas y/o comparaciones.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

El concepto de algoritmo se originó como una forma de registrar procedimientos para resolver problemas matemáticos (ej. al algoritmo de suma, resta, multiplicación y división, o el Algoritmo de Euclides para encontrar el máximo común divisor de dos números). El concepto fue formalizado en 1936 por Alan Turing a través del uso de lo que hoy se llaman *Máquinas de Turing*. Estos conceptos están en la base misma de la Ciencia de la Computación

La mayoría de los algoritmos pueden ser implementados directamente por programas de computación. Otros pueden ser al menos simulados en teoría por programas de computación.

Una restricción que se agrega usualmente a la definición de algoritmo es que éste llegue al estado final en un *número finito de pasos*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

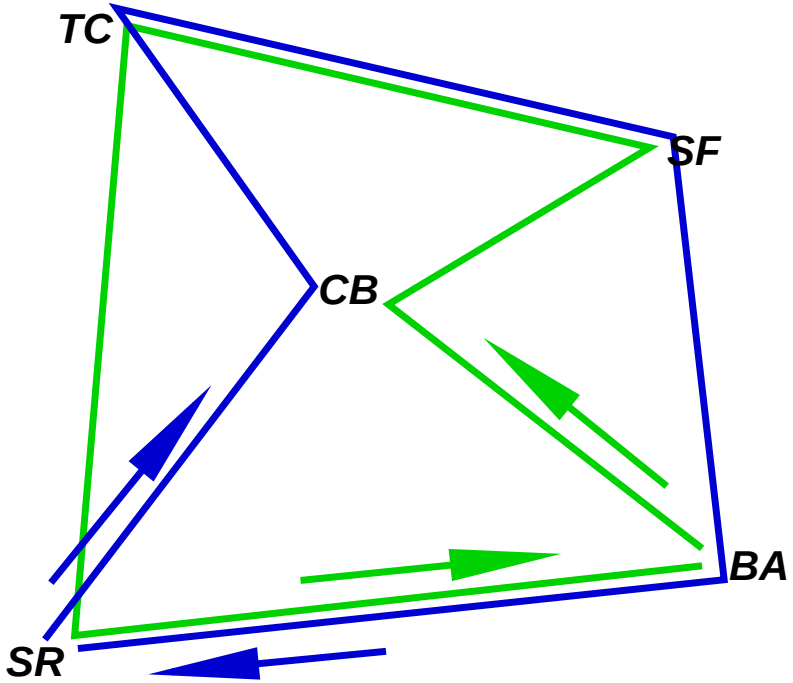
Diseño y análisis de algoritmos (cont.)

- **No existe una regla precisa para escribir un programa que resuelva un problema práctico dado. Al menos por ahora escribir programas es en gran medida un arte.**
- **Sin embargo con el tiempo se han desarrollado un variedad de conceptos que ayudan a desarrollar estrategias para resolver problemas y comparar *a priori* la eficiencia de las mismas.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

- **Por ejemplo supongamos que queremos resolver el *"Problema del Agente Viajero"* (TSP, por *"Traveling Salesman Problem"*) el cual consiste en encontrar el orden en que se debe recorrer un cierto número de ciudades (esto es, una serie de puntos en el plano) en forma de tener un recorrido mínimo.**

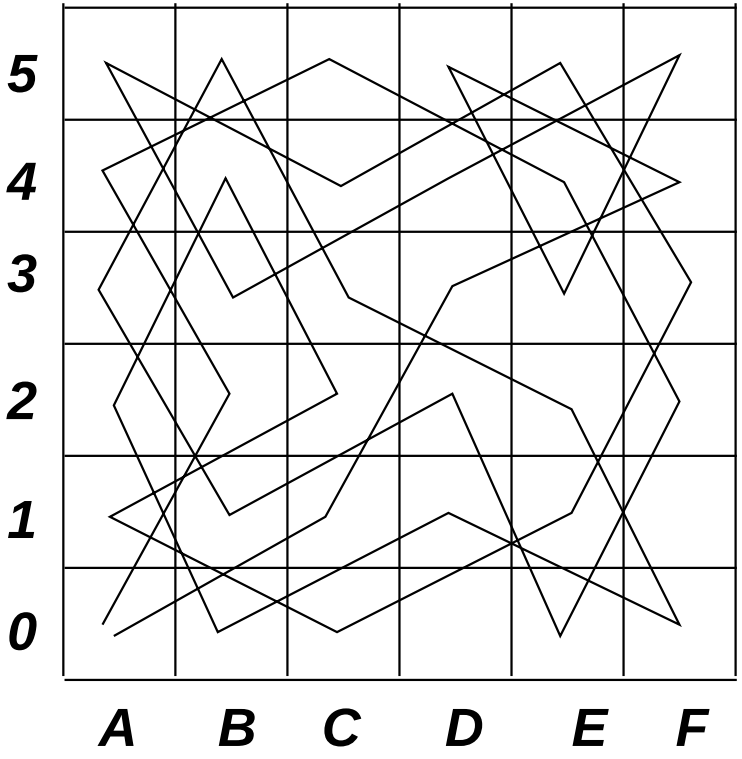


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Este problema surge en una variedad de problemas

- **Aplicaciones prácticas, por ejemplo encontrar caminos mínimos para recorridos de distribución de productos.**
- **Resolver el problema de *"la vuelta del caballo en el tablero de ajedrez"*, es decir, encontrar un camino para el caballo que recorra toda las casillas del tablero pasando una sola vez por cada casilla.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Estrategias

- **Existe una estrategia (trivial) que consiste en evaluar todos los caminos posibles. Pero esta estrategia de *"búsqueda exhaustiva"* tiene un gran defecto, el costo computacional crece de tal manera con el número de ciudades que deja de ser aplicable a partir de una cantidad relativamente pequeña.**
- **Otra estrategia *"heurística"* se basa en buscar un camino que, si bien no es el óptimo (el de menor recorrido sobre todos los posibles) puede ser relativamente bueno en la mayoría de los casos prácticos. Por ejemplo, empezar en una ciudad e ir a la más cercana que no haya sido aún visitada hasta recorrerlas todas.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Algoritmo

Una forma abstracta de plantear una estrategia es en la forma de un *"algoritmo"*, es decir una secuencia de instrucciones cada una de las cuales representa una tarea bien definida y puede ser llevada a cabo en una cantidad finita de tiempo y recursos computacionales. El algoritmo debe terminar en un número finito de pasos. Un algoritmo puede ser usado como una instrucción en otro algoritmo más complejo.

Entonces, comparando diferentes algoritmos para el TSP entre sí, podemos plantear las siguientes preguntas

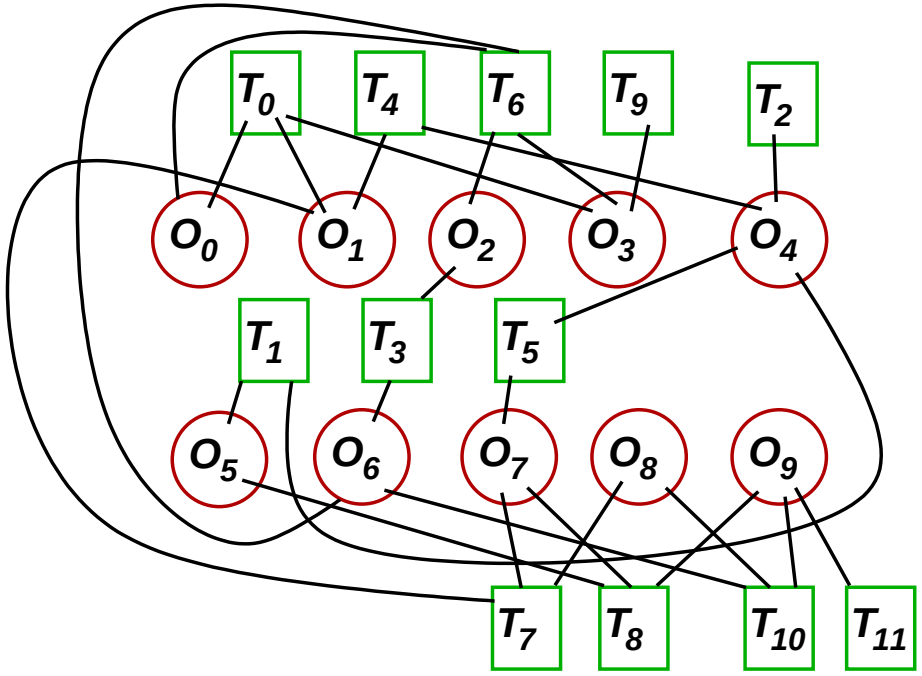
- **¿Da el algoritmo la solución óptima?**
- **Si el algoritmo es iterativo, ¿converge?**
- **¿Cómo crece el esfuerzo computacional a medida que el número de ciudades crece?**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Ejemplo: Sincronización en cálculo distribuido

Consideremos un sistema de procesamiento con varios procesadores que acceden a un área de memoria compartida. En memoria hay una serie de objetos O_0, O_1, ... O_{n-1}, con $n = 10$ y una serie de tareas a realizar T_0, T_1, ... T_{m-1} con $m = 12$. Cada tarea debe modificar un cierto subconjunto de los objetos, según la figura



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Las tareas pueden realizarse en cualquier orden, pero dos tareas *no pueden ejecutarse al mismo tiempo si acceden al mismo objeto*, ya que los cambios hechos por una de ellas puede interferir con los cambios hechos por la otra. Debe entonces desarrollarse un sistema que sincronice entre sí la ejecución de las diferentes tareas.

Una forma trivial de sincronización es ejecutar cada una de las tareas en forma secuencial. Primero la tarea T_0 luego la T_1 y así siguiendo hasta la T_{11}, de esta forma nos aseguramos que no hay conflictos en el acceso a los objetos.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

Sin embargo, esta solución puede estar muy lejos de ser óptima en cuanto al tiempo de ejecución ya que por ejemplo T_0 y T_1 pueden ejecutarse al mismo tiempo en diferentes procesadores ya que modifican diferentes objetos. Si asumimos, para simplificar, que todas las tareas llevan el mismo tiempo de ejecución τ, entonces la versión trivial consume una cantidad de tiempo $m\tau$, mientras que ejecutando las tareas T_0 y T_1 al mismo tiempo reducimos el tiempo de ejecución a $(m-1)\tau$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

El algoritmo a desarrollar debe *"particionar"* las tareas en una serie de p *"etapas"* E_0, ... E_p. Las etapas son simplemente subconjuntos de las tareas y la partición debe satisfacer las siguientes restricciones

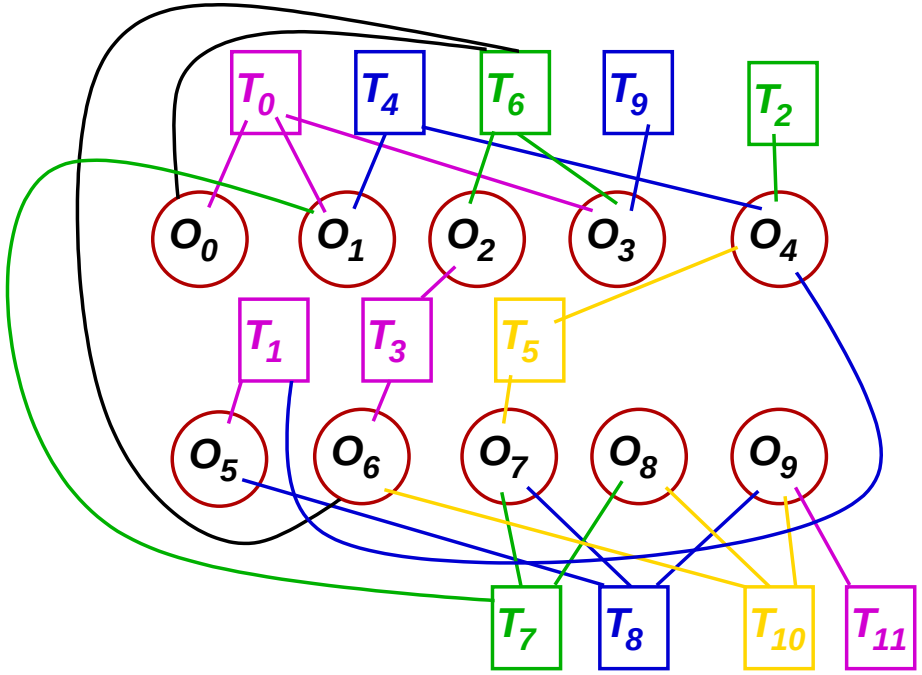
- **Cada tarea debe estar en una y sólo una etapa. (De lo contrario la tarea no se realizaría o se realizaría más de una vez, lo cual es redundante. En el lenguaje de la teoría de conjuntos, estamos diciendo que debemos particionar el conjunto de etapas en un cierto número de subconjuntos *"disjuntos"*.)**
- **Las tareas a ejecutarse en una dada etapa no deben acceder al mismo objeto.**

Una partición *"admisible"* es aquella que satisface todas estas condiciones. El objetivo es *determinar aquella partición admisible que tiene el mínimo número de etapas*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diseño y análisis de algoritmos (cont.)

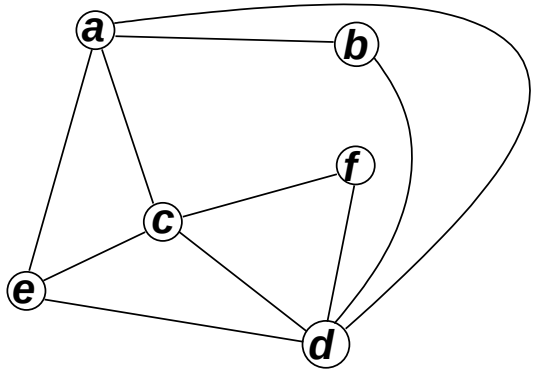
Este problema es muy común, ya que se plantea siempre que hay un número de tareas a hacer y conflictos entre esas tareas, por ejemplo sincronizar una serie de tareas con maquinaria a realizar en una industria, evitando conflictos en el uso del instrumental o maquinaria, es decir no agendar dos tareas para realizar simultáneamente si van a usar el microscopio electrónico.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Introducción básica a grafos

El problema se puede plantear usando una estructura matemática conocida como *"grafo"*. La base del grafo es un conjunto finito V de puntos llamados *"vértices"*. La estructura del grafo está dada por las conexiones entre los vértices. Si dos vértices están conectados se dibuja una línea que va desde un vértice al otro. Estas conexiones se llaman *"aristas"* (*"edges"*) del grafo. Los vértices pueden identificarse con un número de 0 a $n_v - 1$ donde n_v es el número total de vértices.

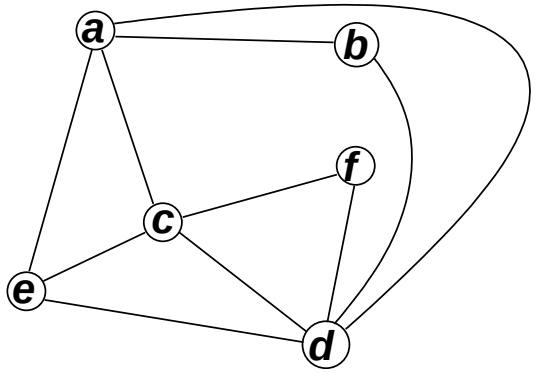


	a	b	c	d	e	f
a	0	1	1	1	1	0
b	1	0	0	1	0	0
c	1	0	0	1	1	1
d	1	1	1	0	1	1
e	1	0	1	1	0	0
f	0	0	1	1	0	0

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Introducción básica a grafos (cont.)

Puede representarse como una matriz A simétrica de tamaño $n_v \times n_v$ con 0's y 1's. Si hay una arista entre el vértice i y el j entonces el elemento A_{ij} es uno, y sino es cero. Además, si existe una arista entre dos vértices i y j entonces decimos que i es *"adyacente"* a j.



	a	b	c	d	e	f
a	0	1	1	1	1	0
b	1	0	0	1	0	0
c	1	0	0	1	1	1
d	1	1	1	0	1	1
e	1	0	1	1	0	0
f	0	0	1	1	0	0

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

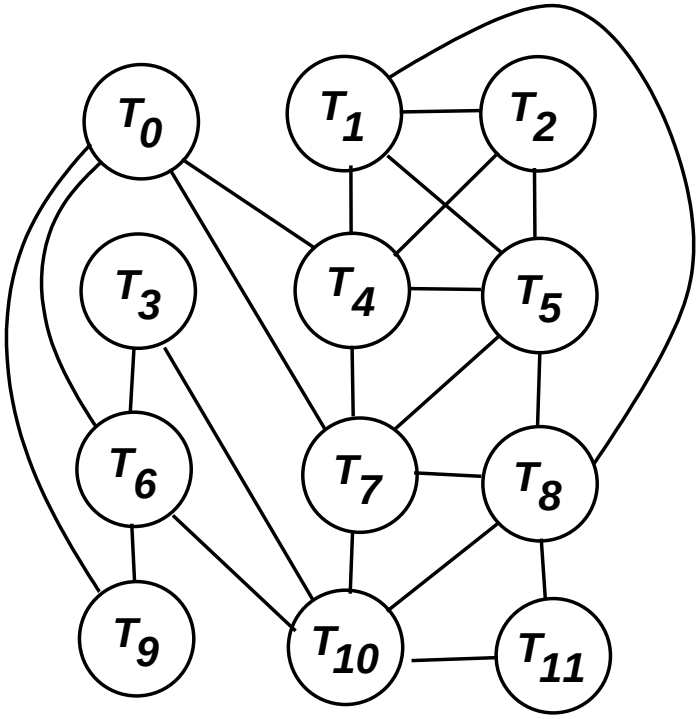
Introducción básica a grafos (cont.)

Para este ejemplo usaremos *"grafos no orientados"*, es decir que si el vértice i está conectado con el j entonces el j está conectado con el i. También existen *"grafos orientados"* donde las aristas se representan por flechas.

Se puede también agregar un peso (un número real) a los vértices o aristas del grafo. Este peso puede representar, por ejemplo, un costo computacional.

Planteo del problema mediante grafos

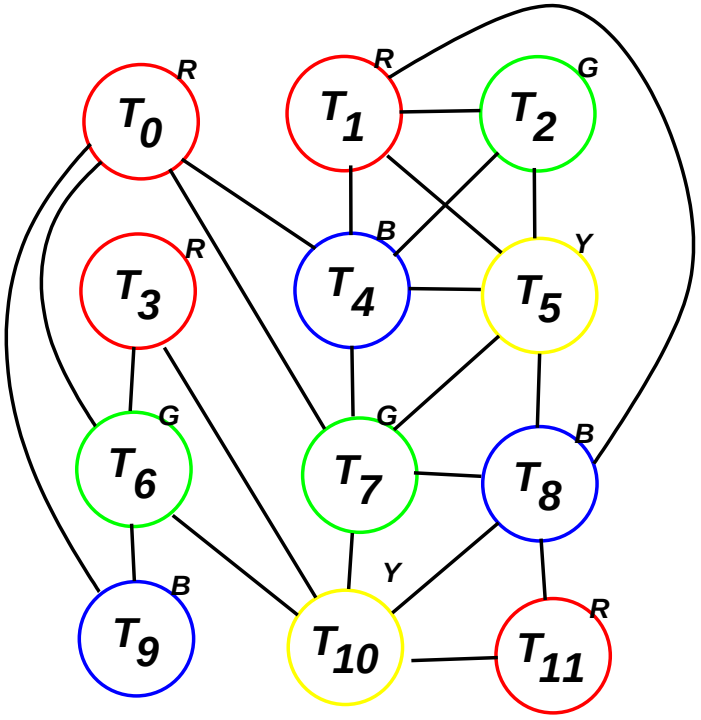
Podemos plantear el problema dibujando un grafo donde los vértices corresponden a las tareas y dibujaremos una arista entre dos tareas si son incompatibles entre sí (modifican el mismo objeto).



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Planteo del problema mediante grafos (cont.)

La buena noticia es que nuestro problema de particionar el grafo ha sido muy estudiado en la teoría de grafos y se llama el problema de *"colorear"* el grafo, es decir se representan gráficamente las etapas asignándole colores a los vértices del grafo. La mala noticia es que se ha encontrado que obtener el coloreado óptimo (es decir el coloreado admisible con la menor cantidad de colores posibles) resulta ser un problema extremadamente costoso en cuanto a tiempo de cálculo.

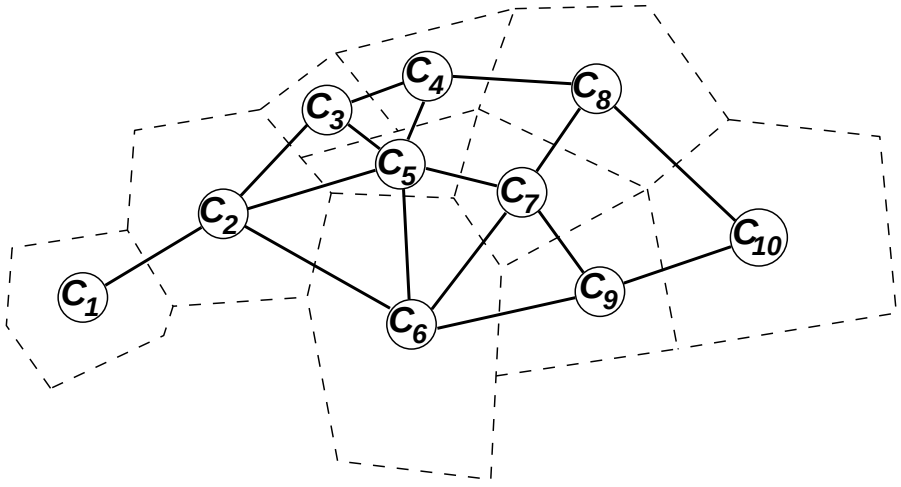


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Planteo del problema mediante grafos (cont.)

El término *"colorear grafos"* viene de un problema que también se puede poner en términos de colorear grafos y es el de colorear países en un mapa.

Debemos asignar a cada país un color, de manera que países limítrofes (esto es, que comparten una porción de frontera de medida no nula) tengan diferentes colores y, por supuesto, debemos tratar de usar el mínimo número de colores posibles.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmo de búsqueda exhaustiva

Probar si el grafo se puede colorear con 1 solo color (esto sólo es posible si no hay ninguna arista en el grafo). Si esto es posible el problema está resuelto (no puede haber coloraciones con menos de un color). Si no es posible entonces generamos todas las coloraciones con 2 colores, para cada una de ellas verificamos si satisface las restricciones o no, es decir si es admisible. Si lo es, el problema está resuelto: encontramos una coloración admisible con dos colores y ya verificamos que con 1 solo color no es posible. Si no encontramos ninguna coloración admisible de 2 colores entonces probamos con las de 3 colores y así sucesivamente. Si encontramos una coloración de n_c colores entonces será óptima, ya que previamente verificamos para cada número de colores entre 1 y $n_c - 1$ que no había ninguna coloración admisible.

Algoritmo de búsqueda exhaustiva (cont.)

Ahora tratando de responder las preguntas que planteamos para un algoritmo genérico

- **Da la solución óptima.**
- **Termina en un número finito de pasos ya que a lo sumo puede haber $n_c = n_v$ colores, es decir la coloración que consiste en asignar a cada vértice un color diferente es siempre admisible.**

Generación de las coloraciones

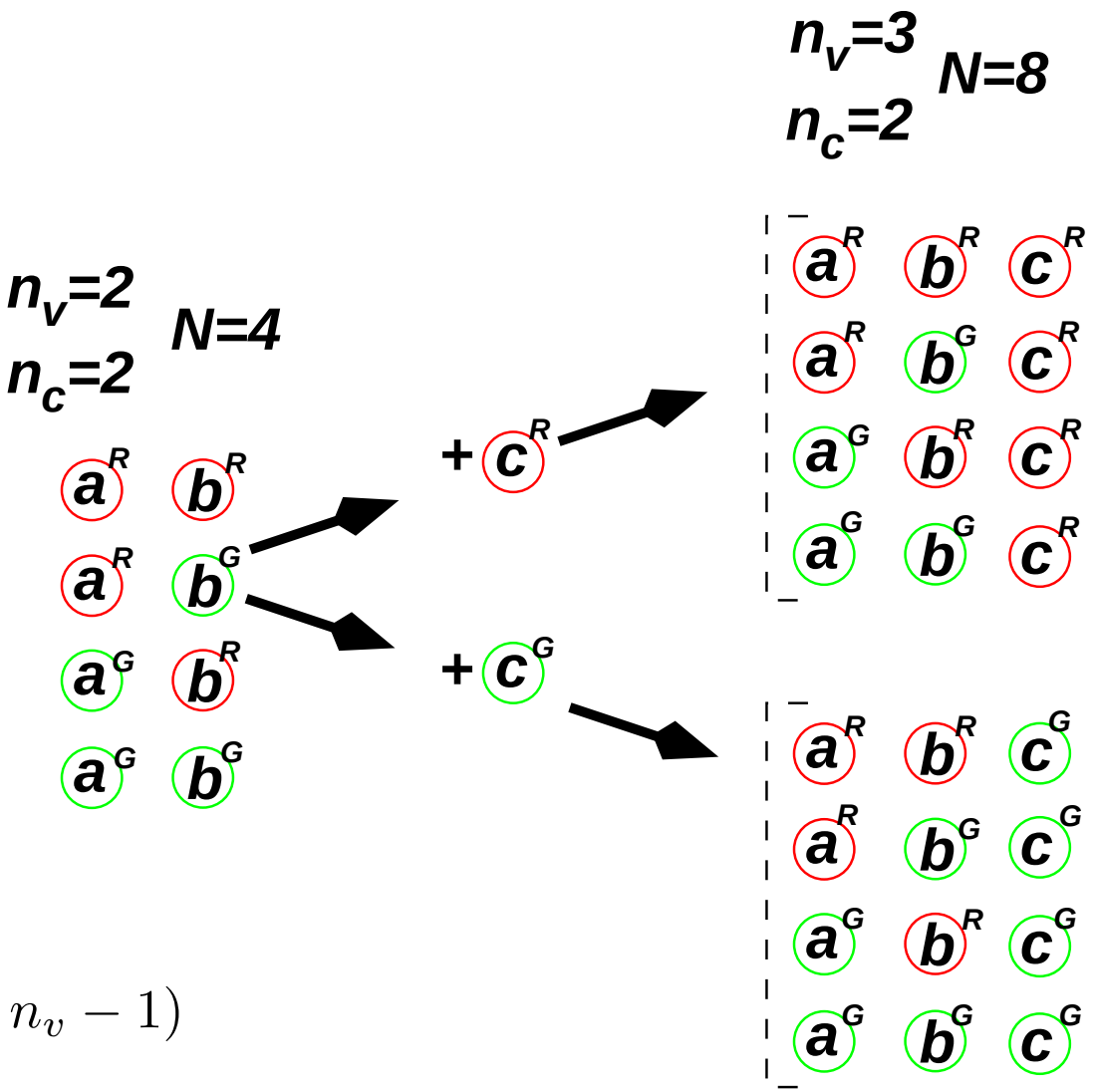
En realidad todavía falta resolver un punto del algoritmo y es cómo generar todas las coloraciones posibles de n_c colores. Además esta parte del algoritmo debe ser ejecutable en un número finito de pasos así que trataremos de evaluar cuantas coloraciones $N(n_c, n_v)$ hay para n_v vértices con n_c colores. Notemos primero que el procedimiento para generar las coloraciones es independiente de la estructura del grafo (es decir de las aristas), sólo depende de cuantos vértices hay en el grafo y del número de colores que pueden tener las coloraciones.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de las coloraciones (cont.)

Para $n_c = 1$ es trivial, hay una sola coloración donde todos los vértices tienen el mismo color, es decir $N(n_c = 1, n_v) = 1$ para cualquier n_v.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))



Coloraciones de 3 objetos con 2 colores

$$N(2,3) = 2\,N(2,2)$$

$$N(n_c, n_v) = n_c\,N(n_c, n_v - 1)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de las coloraciones (cont.)

Recursivamente, para cualquier $n_c, n_v \geq 1$, tenemos que

$$\begin{aligned} N(n_c, n_v) &= n_c\, N(n_c, n_v - 1) \\ &= n_c^2\, N(n_c, n_v - 2) \\ &\vdots \\ &= n_c^{n_v - 1}\, N(n_c, 1) \end{aligned} \tag{1}$$

Pero el número de coloraciones para un sólo vértice con n_c colores es n_c, de manera que

$$N(n_c, n_v) = n_c^{n_v} \tag{2}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de las coloraciones (cont.)

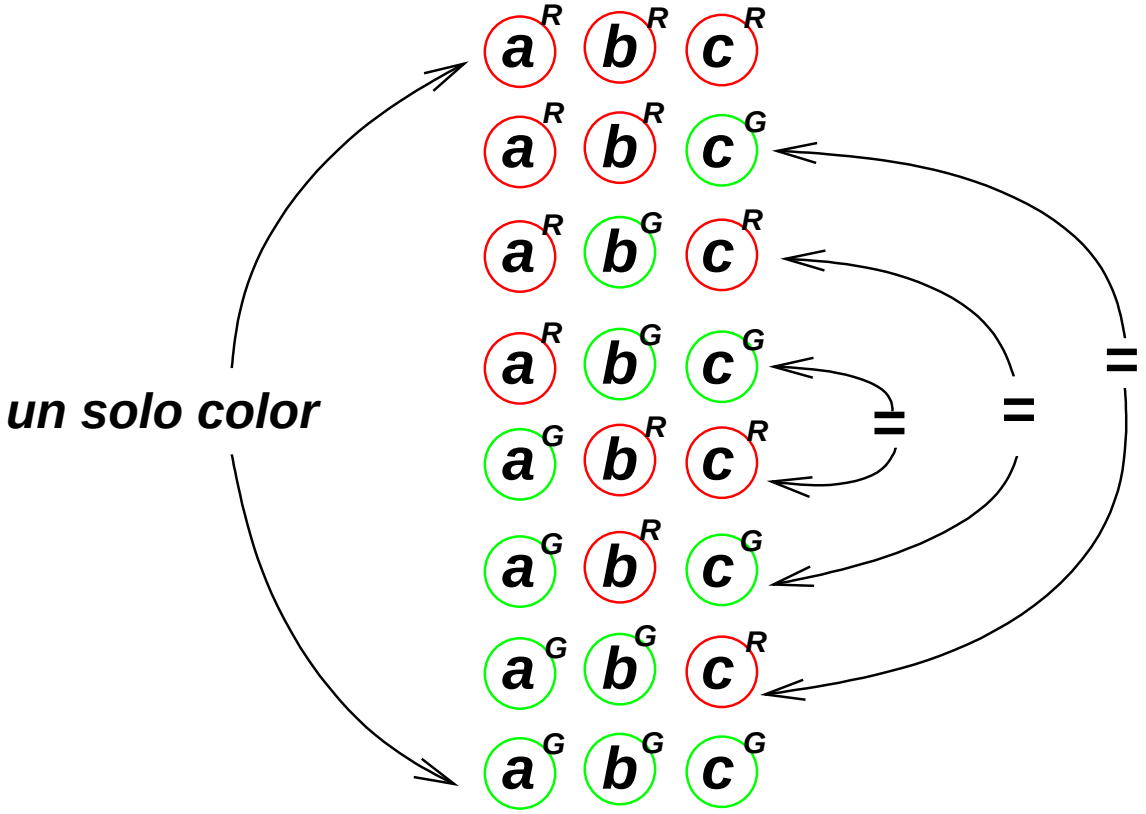
Esto cierra con la última pregunta, ya que vemos que el número de pasos para cada uno de los colores es finito, y hay a lo sumo n_v colores de manera que el número total de posibles coloraciones a verificar es finito. Notar de paso que esta forma de contar las coloraciones es también *"constructiva"*, da un procedimiento para generar todas las coloraciones, si uno estuviera decidido a implementar la estrategia de búsqueda exhaustiva.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de las coloraciones (cont.)

Notemos que el número de coloraciones en realidad está *"estimado por exceso"*.

- **Algunas coloraciones en realidad tienen menos de 2 colores.**
- **Otras en realidad son esencialmente la misma coloración, ya que corresponden a intercambiar colores entre sí (por ejemplo rojo con verde).**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Crecimiento del tiempo de ejecución

No consideremos, por ahora, la eliminación de coloraciones redundantes y consideremos que para aplicar la estrategia exhaustiva al problema de coloración de un grafo debemos evaluar $N = n_v^{n_v}$ coloraciones. Esto corresponde al peor caso de que el grafo necesite el número máximo de $n_c = n_v$ colores.

Para verificar si una coloración dada es admisible debemos realizar un cierto número de operaciones. En el peor de los casos el número de operaciones necesario para verificar una dada coloración es igual al número de aristas y a lo sumo el número de aristas es $n_v \cdot n_v$ (el número de elementos en la forma matricial del grafo) de manera que para verificar todas las coloraciones necesitamos verificar

$$N_{\mathrm{be}} = n_v^2 \, n_v^{n_v} = n_v^{n_v+2}$$

aristas. Asumiendo que el tiempo de verificar una arista es constante, este es el orden del número de operaciones a realizar.

Crecimiento del tiempo de ejecución (cont.)

El crecimiento de la función $n_v^{n_v+2}$ con el número de vértices es tan rápido que hasta puede generar asombro. Consideremos el tiempo que tarda una computadora personal típica en evaluar todas las posibilidades para $n_v = 20$ vértices. Tomando un procesador de 2.4 GHz (un procesador típico al momento de escribir este apunte) y asumiendo que podemos escribir un programa tan eficiente que puede evaluar una arista por cada ciclo del procesador (en la práctica esto es imposible y al menos necesitaremos unas decenas de ciclos para evaluar una coloración) el tiempo en años necesario para evaluar todas las coloraciones es de

$$T = \frac{20^{22}}{2.4 \times 10^9 \,.\, 3600 \,.\, 24 \,.\, 365} = 5.54 \times 10^{11} \text{ años}$$

Esto es unas 40 veces la edad del universo (estimada en 15.000.000.000 de años).

Crecimiento del tiempo de ejecución (cont.)

Algo que debe quedar en claro es que el problema no esta en la velocidad de las computadoras, sino en la estrategia de búsqueda exhaustiva. Incluso haciendo uso de las más sofisticadas técnicas de procesamiento actuales los tiempos no bajarían lo suficiente. Por ejemplo usando uno de los *"clusters"* de procesadores más grandes existentes actualmente (con más de mil procesadores, ver `http://www.top500.org`) sólo podríamos bajar el tiempo de cálculo al orden de los millones de años.

Otra forma de ver el problema es preguntarse cuál es el máximo número de vértices que se puede resolver en un determinado tiempo, digamos una hora de cálculo. La respuesta es que ya con $n_v = 15$ se tienen tiempos de más de 5 horas.

En la sección siguiente veremos que si bien la eliminación de las coloraciones redundantes puede reducir significativamente el número de coloraciones a evaluar, el crecimiento de la función sigue siendo similar y no permite pasar de unas cuantas decenas de vértices.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Búsqueda exhaustiva mejorada

Eliminando las coloraciones redundantes se obtiene una gran reducción en el número de coloraciones a evaluar. $N_{\text{bem}} \approx n_v^{n_v/2+2}$**. Esto significa una gran mejora con respecto a** $n_v^{n_v}$**, para 20 vértices pasamos de** 4.2×10^{28} **coloraciones a** 5.2×10^{13} **coloraciones y el tiempo de cómputo se reduce a sólo 99 días. Está claro que todavía resulta ser excesivo para un uso práctico.**

n_v	coloraciones	coloraciones diferentes
1	1	1
2	4	2
3	27	5
4	256	15
5	3125	52
6	46656	203
7	823543	877

Algoritmo heurístico ávido

Una estrategia diferente a la de búsqueda exhaustiva es la de buscar una solución que, si bien no es la óptima (es decir, la mejor de todas), sea aceptablemente buena, y se pueda obtener en un tiempo razonable.

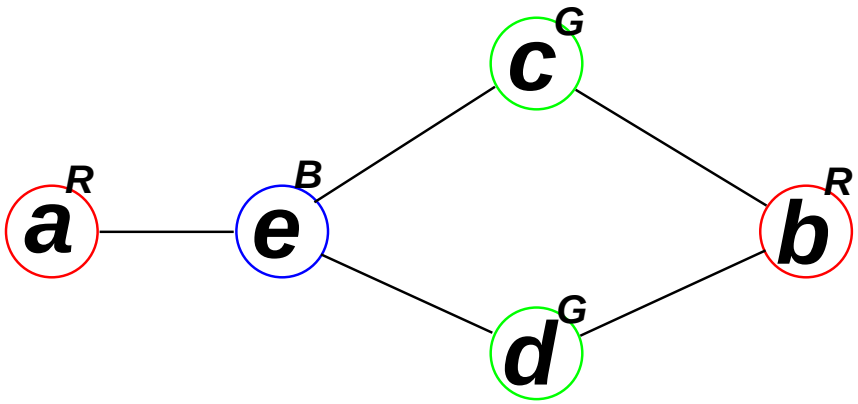
Un algoritmo que produce una solución razonablemente buena haciendo hipótesis *"razonables"* se llama *"heurístico"*. Del diccionario: *heurístico: una regla o conjunto de reglas para incrementar la posibilidad de resolver un dado problema*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmo heurístico ávido (cont.)

Un posible algoritmo heurístico para colorear grafos es el siguiente algoritmo *"ávido"*.

- **Seleccionar algún vértice no coloreado y asignarle el nuevo color.**
- **Recorrer la lista de vértices no coloreados. Para cada vértice no coloreado determinar si está conectado (esto es, posee algún vértice en común) con un vértice del nuevo color.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

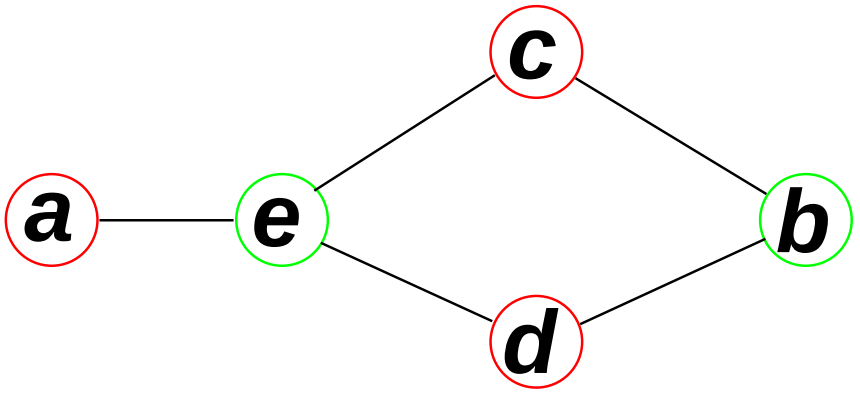
Algoritmo heurístico ávido (cont.)

Esta aproximación es llamada ávida ya que asigna colores tan rápido como lo puede hacer, sin tener en cuenta las posibles consecuencias negativas de tal acción. Si estuviéramos escribiendo un programa para jugar al ajedrez, entonces una estrategia ávida, sería evaluar todas las posibles jugadas y elegir la que da la mejor ventaja material. En realidad no se puede catalogar a los algoritmos como ávidos en forma absoluta, sino que se debe hacer en forma comparativa: hay algoritmos más ávidos que otros. En general cuanto más ávido es un algoritmo más simple es y más rápido es en cuanto a avanzar para resolver el problema, pero por otra parte explora en menor medida el espacio de búsqueda y por lo tanto puede dar una solución peor que otro menos ávido. Volviendo al ejemplo del ajedrez, un programa que, además de evaluar la ganancia material de la jugada a realizar, evalúe las posibles consecuencias de la siguiente jugada del oponente requerirá mayor tiempo pero a largo plazo producirá mejores resultados.

Algoritmo heurístico ávido (cont.)

El algoritmo encuentra una solución con tres colores, sin embargo se puede encontrar una solución con dos colores. Esta última es óptima ya que una mejor debería tener sólo un color, pero esto es imposible ya que entonces no podría haber ninguna arista en el grafo. Este ejemplo ilustra perfectamente que si bien el algoritmo ávido da una solución razonable, ésta puede no ser la óptima.

Notemos también que la coloración producida por el algoritmo ávido depende del orden en el que se recorren los vértices. En el caso previo, si recorriéramos los nodos en el orden $\{a, e, c, d, b\}$, obtendríamos la coloración óptima.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico

```
1  void greedy(graph &G, set<int> &no_col,
2              set<int> &nuevo_color,
3              vector<int> &tabla_color,int color) {
4    // Asigna a 'nuevo_color' un conjunto de vertices
5    // de 'G' a los cuales puede darse el mismo nuevo color
6    // sin entrar en conflicto con los ya coloreados
7    nuevo_color.clear();
8    set<int>::iterator q;
9    for (q=no_col.begin(); q!=no_col.end(); q++) {
10     if (/* '*q' no es adyacente a
11            ningun vertice en 'nuevo_color' . . . */) {
12       // marcar a '*q' como coloreado
13       // . . .
14       // agregar '*q' a 'nuevo_color'
15       // . . .
16     }
17   }
18 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

- **Está escrito en *"seudo-código"* (por ejemplo *"elegir un vértice no coloreado"*).**
- **Luego en un paso de *"refinamiento"* posterior estas sentencias en seudo-código son refinadas en tareas más pequeñas, hasta que finalmente terminamos con un código que puede ser compilado y linkeditado en un programa.**
- **Los vértices son identificados por un entero de 0 a `nv-1`.**
- **La rutina además mantiene una tabla `tabla_color` donde finalmente quedarán los colores de cada vértice.**
- **El argumento de entrada `color` indica el color (un entero empezando desde 0 e incrementándose de a 1 en cada llamada a `greedyc`) con el cual se está coloreando en esta llamada a `greedyc`.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

Para verificar si `*q` es adyacente a algún vértice de `nuevo_color` debemos recorrer todos los nodos de `nuevo_color` y verificar si hay alguna arista entre los mismos y `*q`. Es decir, el seudo-código:

```
1    if (/* '*q' no es adyacente a
2          ningun vertice en 'nuevo_color' . . . */) {
```

es refinado a

```
1    int adyacente=0;
2    for (w=nuevo_color.begin();
3         w!=nuevo_color.end(); w++) {
4      if (/* '*w' es adyacente a '*q' . . . */) {
5        adyacente = 1;
6        break;
7      }
8    }
9    if (!adyacente) {
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

Para refinar el condicional necesitamos definir la clase `grafo`, para ello utilizaremos una representación muy simple, útil para grafos pequeños basada en mantener una tabla de unos y ceros. Los elementos de la matriz se acceden a través de una función miembro `edge(j,k)` que retorna una referencia al elemento correspondiente de la matriz. `edge()` retorna una referencia al elemento correspondiente, puede usarse tanto para insertar aristas en el grafo

```
1 G.edge(j,k) = 1;
```

como para consultar si un dado par de vértices es adyacente o no

```
1 if (!G.edge(j,k)) {
2   // no estan conectados
3   // . . .
4 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

Con esta interfase para grafos, podemos definir completamente el `if`. El código final es

```
1  void greedyc(graph &G, set<int> &no_col,
2               set<int> &nuevo_color,
3               vector<int> &tabla_color,int color) {
4    // Asigna a 'nuevo_color' un conjunto de vertices
5    // de 'G' a los cuales puede darse el mismo nuevo color
6    // sin entrar en conflicto con los ya coloreados
7    nuevo_color.clear();
8    set<int>::iterator q,w;
9    for (q=no_col.begin(); q!=no_col.end(); q++) {
10     int adyacente=0;
11     for (w=nuevo_color.begin();
12          w!=nuevo_color.end(); w++) {
13       if (G.edge(*q,*w)) {
14         adyacente = 1;
15         break;
16       }
17     }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
18      if (!adyacente) {
19        // marcar a '*q' como coloreado
20        tabla_color[*q] = color;
21        // agregar '*q' a 'nuevo_color'
22        nuevo_color.insert(*q);
23      }
24    }
25  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

```
1  void greedy(graph &G, int nv,
2              vector<int> &tabla_color) {
3    int color=0;
4    set<int> nuevo_color, no_col;
5    set<int>::iterator q;
6    // Inicialmente ponemos todos los vertices en
7    // 'no_col'
8    for (int k=0; k<nv; k++) no_col.insert(k);
9    while (1) {
10     // Determina a cuales vertices podemos asignar el
11     // nuevo color
12     greedyc(G,no_col,nuevo_color,
13             tabla_color,color);
14     // Saca los vertices que se acaban de colorear
15     // ('nuevo_color') de 'no_col'
16     // ...
17     // Detecta el fin del algoritmo cuando ya no hay
18     // mas vertices para colorear.
19     // ...
20     color++;
21   }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

```
1  void greedy(graph &G, int nv,
2              vector<int> &tabla_color) {
3    int color=0;
4    set<int> nuevo_color, no_col;
5    set<int>::iterator q;
6    // Inicialmente ponemos todos los vertices en 'no_col'
7    for (int k=0; k<nv; k++) no_col.insert(k);
8    while (1) {
9      // Determina a cuales vertices podemos asignar
10     // el nuevo color
11     greedyc(G,no_col,nuevo_color,tabla_color,color);
12     // Saca los vertices que se acaban de colorear
13     // ('nuevo_color') de 'no_col'
14     for (q=nuevo_color.begin();
15          q!=nuevo_color.end(); q++)
16       no_col.erase(*q);
17     // Detecta el fin del algoritmo cuando ya no hay
18     // mas vertices para colorear.
19     if (!no_col.size()) return;
20     color++;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
21      }
22  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del algoritmo heurístico (cont.)

Crecimiento del tiempo de ejecución para el algoritmo ávido

El tiempo de ejecución para este algoritmo resulta ser $\propto n_v^3$. De esta forma se pueden colorear grafos de 2000 vértices en 40 secs.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa

- **La eficiencia de un código va en forma inversa con la cantidad de recursos que consume, principalmente tiempo de CPU y memoria. A veces en programación la eficiencia se contrapone con la sencillez y legibilidad de un código.**
- **Sin embargo en ciertas aplicaciones la eficiencia es un factor importante que no podemos dejar de tener en cuenta. Si escribimos un programa para buscar un nombre en una agenda personal de 200 registros, entonces probablemente la eficiencia no es la mayor preocupación. Pero si escribimos un algoritmo para un motor de búsqueda en un número de entradas $> 10^9$, como es común en las aplicaciones para buscadores en Internet hoy en día, entonces la eficiencia probablemente pase a ser un concepto fundamental. Para tal volumen de datos, pasar de un algoritmo $O(n \log n)$ a uno $O(n^{1.3})$ puede ser fatal.**

Tiempos de ejecución de un programa (cont.)

- **Más importante que saber escribir programas eficientemente es saber *cuándo* y *dónde* preocuparse por la eficiencia.**
- **Un programa está compuesto en general por varios componentes o módulos. No tiene sentido preocuparse por la eficiencia de un módulo dado si éste representa un 5% del tiempo total de cálculo.**
- **En tal módulo tal vez sea mejor preocuparse por la robustez y sencillez de programación que por la eficiencia.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa (cont.)

Para fijar ideas, pensemos en un programa que ordena de menor a mayor una serie de números enteros. El tiempo de ejecución de un programa depende de:

- ***La eficiencia del compilador y las opciones de optimización que pasamos al mismo en tiempo de compilación.***
- ***El tipo de instrucciones y la velocidad del procesador donde se ejecutan las instrucciones compiladas.***
- ***Los datos del programa.*** **En el ejemplo, la cantidad de números y su distribución estadística: ¿son todos iguales?, ¿están ya ordenados o casi ordenados?**
- ***La "complejidad algorítmica" del algoritmo subyacente*****. ¿Es el algoritmo**
  - ▷ $O(n \log n)$ **(*"quicksort"*), o**
  - ▷ $O(n^2)$ **(burbuja, selección ...)?**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa (cont.)

- **En muchos casos, el tiempo de ejecución depende de alguna *"medida"* del tamaño de los datos.**
- **Por ejemplo sumar un arreglo de n números no depende de los números en sí mismos sino de la longitud n del arreglo.**

$$T(n) = cn \tag{3}$$

- **c es una constante que representa el tiempo necesario para sumar un elemento.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa (cont.)

```
1 int search(int l,int *a,int n) {
2   int j;
3   for (j=0; j<n; j++)
4     if (a[j]==l) break;
5   return j;
6 }
```

- En otros casos, si bien el tiempo de ejecución sí depende de los datos específicos, *en promedio* sólo depende del tamaño de los datos.
- Buscar la ubicación de un elemento l en un arreglo a. El tiempo de ejecución dependerá fuertemente de la ubicación del elemento dentro del arreglo, y por lo tanto es proporcional a la posición j del elemento dentro del vector.
- El mejor caso es cuando el elemento está al principio del arreglo, mientras que el peor es cuando está al final o cuando no está.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución de un programa (cont.)

- ***"En promedio"*** **el elemento buscado estará en la zona media del arreglo, de manera que una ecuación como la $T(n) = cn$ será válida (en promedio).**

$$\begin{aligned} T(n) &= cj \\ T_{\text{peor}}(n) &= cn \\ T_{\text{prom}}(n) &= c\frac{n}{2} \end{aligned} \tag{4}$$

- **En general, determinar analíticamente el tiempo de ejecución de un algoritmo puede ser difícil. Muchas veces, encontrar el $T_{\text{peor}}(n)$ es una tarea relativamente más fácil. Determinar el $T_{\text{prom}}(n)$ puede a veces ser más fácil y otras veces más difícil.**

Notación asintótica

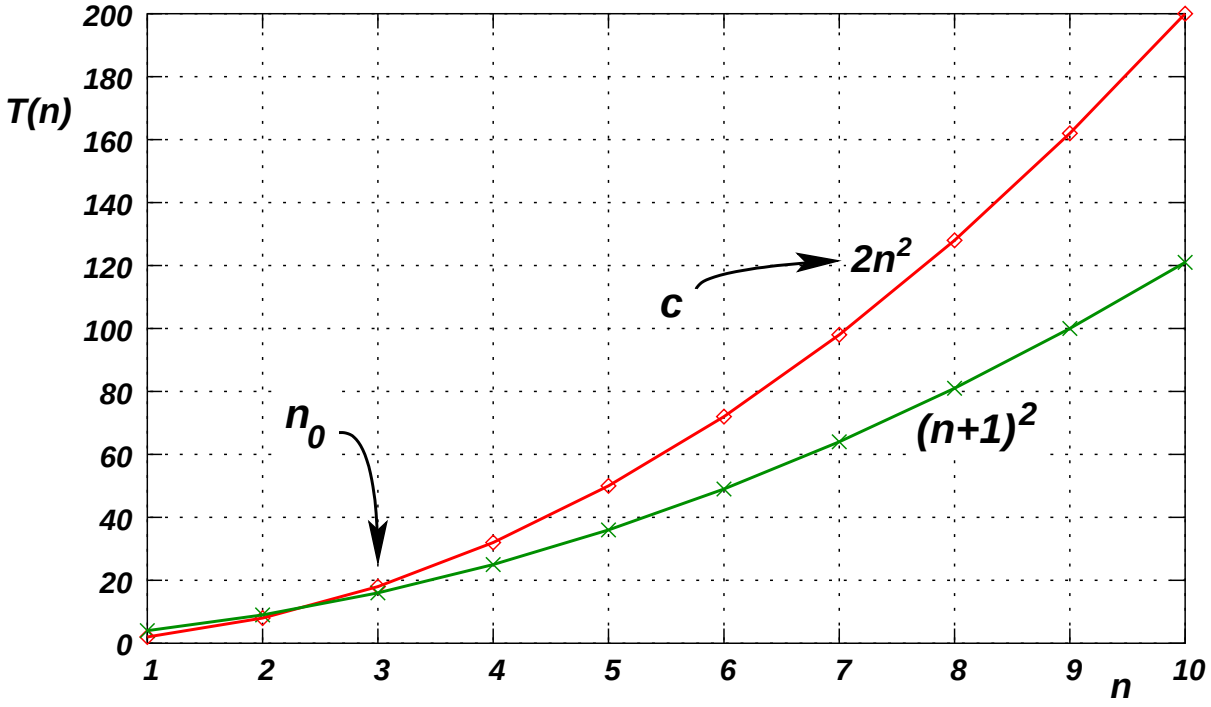
Para poder obtener una rápida comparación entre diferentes algoritmos usaremos la *"notación asintótica"* $O(...)$. Por ejemplo, decimos que el tiempo de ejecución de un programa es $T(n) = O(n^2)$ (se lee *"$T(n)$ es orden n^2"*) si existen constantes $c, n_0 > 0$ tales que

$$T(n) \leq c\, n^2, \quad \textbf{para} \quad n \geq n_0 \tag{5}$$

La idea es que no nos interesa cómo se comporta la función $T(n)$ para valores de n pequeños sino sólo la tendencia para $n \to \infty$.

Notación asintótica (cont.)

Ejemplo: **Sea $T(n) = (n+1)^2$ vemos gráficamente que la relación $T(n) \leq 2n^2$ es cierta para valores de $3 \leq n \leq 10$. O sea que *parece* ser válido para $n_0 = 3$ y $c = 2$.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Para ver que esta relación es válida para *todos* los valores de n tales que $n \geq 3$, entonces debemos recurrir a un poco de álgebra.

$$
\begin{aligned}
n &\geq 3, \\
n-1 &\geq 2, \\
(n-1)^2 &\geq 4, \\
n^2 - 2n + 1 &\geq 4, \\
n^2 &\geq 3 + 2n, \\
2n^2 &\geq 3 + 2n + n^2 \\
2n^2 &\geq (n+1)^2 + 2 \geq (n+1)^2
\end{aligned}
\tag{6}
$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

La notación $O(...)$ puede usarse con otras funciones, es decir $O(n^3)$, $O(2^n)$, $O(\log n)$. En general decimos que $T(n) = O(f(n))$ *(se lee "$T(n)$ es orden $f(n)$") si existen constantes $c, n_0 > 0$ tales que*

$$T(n) \leq c\, f(n), \quad \textbf{para} \quad n \geq n_0 \tag{7}$$

Se suele llamar a $f(n)$ la *"tasa de crecimiento"* de $T(n)$.

Notación asintótica (cont.)

Invariancia ante constantes multiplicativas

Podemos ver que la definición de la tasa de crecimiento es invariante ante constantes multiplicativas, es decir

$$T(n) = O(c\,f(n)) \quad \Longrightarrow \quad T(n) = O(f(n)) \tag{8}$$

Por ejemplo, si $T(n) = O(2n^3)$ entonces también es $O(n^3)$.

Demostración: **Esto puede verse facilmente ya que si $T(n) = O(2n^3)$, entonces existen c y n_0 tales que $T(n) \leq c\,2n^3$ para $n \geq n_0$, pero entonces también podemos decir que $T(n) \leq c'n^3$ para $n \geq n_0$, con $c' = 2c$.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Invariancia de la tasa de crecimiento ante valores en un conjunto finito de puntos

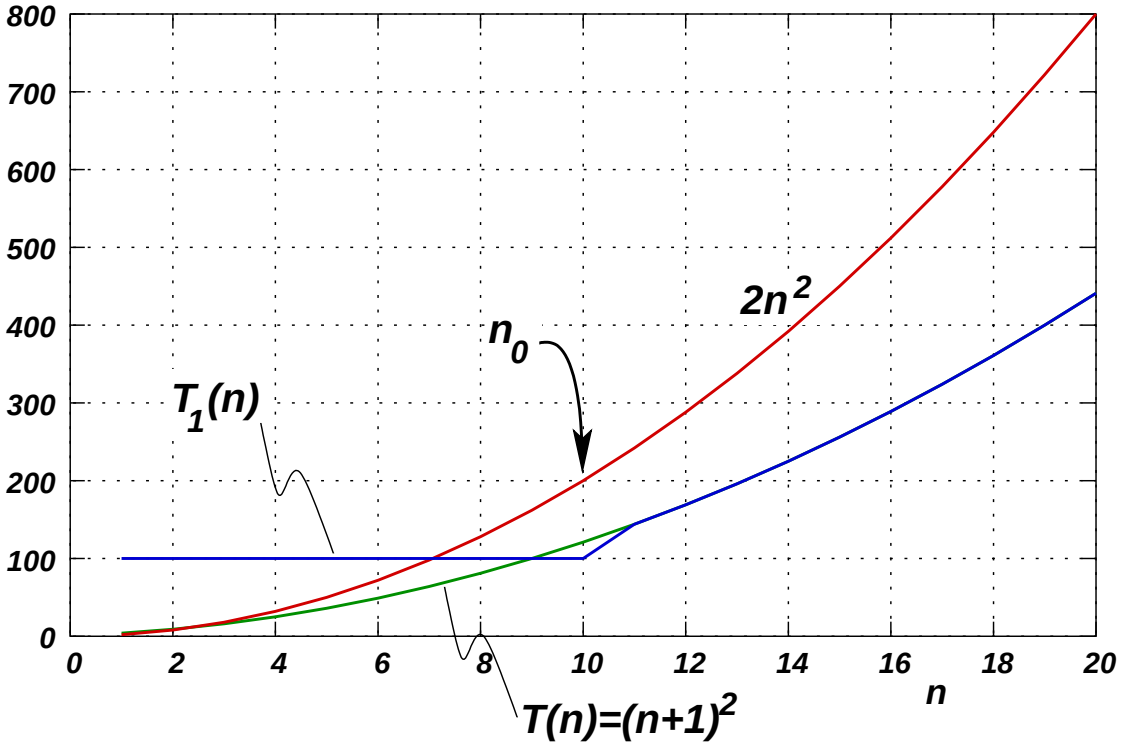
Es decir si

$$T_1(n) = \begin{cases} 100 & ; \textbf{ para } n < 10, \\ (n+1)^2 & ; \textbf{ para } n \geq 10, \end{cases} \tag{9}$$

entonces vemos que $T_1(n)$ coincide con la función $T(n) = (n+1)^2$ estudiada antes.

Notación asintótica (cont.)

$T(n)$ y $T_1(n)$ **sólo difieren en un número finito de puntos (los valores de** $n < 10$**)**



Demostración: **Esto puede verse ya que si** $T(n) < 2n^2$ **para** $n \geq 3$ **(como se vio en el ejemplo citado), entonces** $T_1(n) < 2n^2$ **para** $n > n_0' = 10$**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Transitividad

La propiedad $O(\ )$ es transitiva, es decir si $T(n) = O(f(n))$ y $f(n) = O(g(n))$ entonces $T(n) = O(g(n))$.

Demostración: **si $T(n) \leq cf(n)$ para $n \geq n_0$ y $f(n) \leq c'\, g(n)$ para $n \geq n'_0$, entonces $T(n) \leq c''\, g(n)$ para $n \geq n''_0$, donde $c'' = cc'$ y $n''_0 = \max(n_0, n'_0)$. En cierta forma, $O(...)$ representa una relación de orden entre las funciones (como "$<$" entre los números reales).**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Regla de la suma

Si $f(n) = O(g(n))$, con a, b son constantes positivas, entonces $a\,f(n) + b\,g(n) = O(g(n))$. Es decir, si en una expresión tenemos una serie de términos, sólo queda el *"mayor"* de ellos (en el sentido de $O(...)$). Así por ejemplo, si $T(n) = 2n^3 + 3n^5$, entonces puede verse fácilmente que $n^3 = O(n^5)$ por lo tanto, $T(n) = O(n^5)$.

Demostración: **Si $f(n) \leq cg(n)$ para $n \geq n_0$ entonces $a\,f(n) + b\,g(n) \leq c'g(n)$ para $n \geq n_0$ con $c' = ac + b$.**

Notación asintótica (cont.)

Regla del producto

Si $T_1(n) = O(f_1(n))$ **y** $T_2(n) = O(f_2(n))$ **entonces** $T_1(n)\,T_2(n) = O(f_1(n)\,f_2(n))$**.**

Demostración: **Si** $T_1(n) \leq c_1 f_1(n)$ **para** $n \geq n_{01}$ **y** $T_2(n) \leq c_2 f_2(n)$ **para** $n \geq n_{02}$ **entonces** $T_1(n)T_2(n) \leq c\, f_1(n)\, f_2(n)$ **para** $n > n_0 = \max(n_{01}, n_{02})$ **con** $c = c_1 c_2$**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Funciones típicas utilizadas en la notación asintótica

Cualquier función puede ser utilizada como tasa de crecimiento, pero las más usuales son, en orden de crecimiento

$$1 < \log n < \sqrt{n} < n < n^2 < \ldots < n^p < 2^n < 3^n < \cdots < n! < n^n \quad (10)$$

- **Aquí "$<$" representa $O(\ldots)$. Es decir, $1 = O(\log n)$, $\log n = O(\sqrt{n})$, ...**
- **La función logaritmo crece menos que cualquier potencia n^α con $\alpha > 1$.**
- **Los logaritmos en diferente base son equivalentes entre sí por la bien conocida relación**

$$\log_b n = \log_b a \, \log_a n, \quad (11)$$

de manera que en muchas expresiones con logaritmos no es importante la base utilizada. En computación científica es muy común que aparezcan expresiones con logaritmos en base 2.

Notación asintótica (cont.)

Funciones típicas utilizadas en la notación asintótica

- 1 **representa las funciones constantes.**
- **Las potencias** n^α **(con** $\alpha > 0$**) se comparan entre sí según sus exponentes, es decir** $n^\alpha = O(n^\beta)$ **si** $\alpha \leq \beta$**. Por ejemplo,** $n^2 = O(n^3)$**,** $n^{1/2} = O(n^{2/3})$**.**
- **Las exponenciales** a^n **(con** $a > 1$**) se comparan según su base, es decir que** $a^n = O(b^n)$ **si** $a \leq b$**. Por ejemplo** $2^n = O(3^n)$**. En los problemas de computación científica es muy común que aparezcan expresiones con base** $a = 2$**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

La función factorial

Una función que aparece muy comúnmente en problemas combinatorios es la función factorial $n!$. Para poder comparar esta función con otras para grandes valores de n es conveniente usar la *"aproximación de Stirling"*

$$n! \sim \sqrt{2\pi}\, n^{n+1/2}\, \mathrm{e}^{-n} \tag{12}$$

Gracias a esta aproximación es fácil ver que

$$n! = O(n^n) \tag{13}$$

y

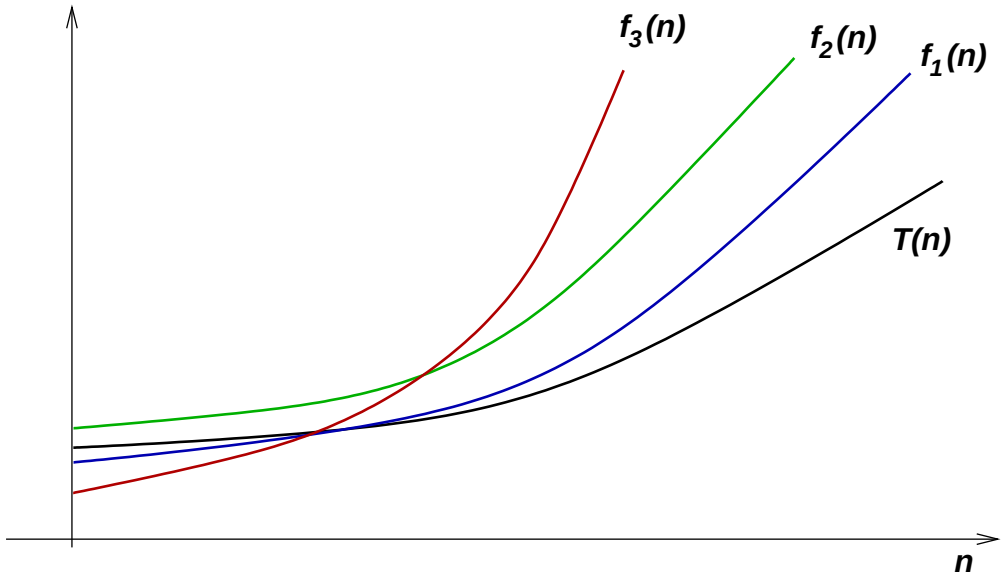
$$a^n = O(n!), \tag{14}$$

Notación asintótica (cont.)

¿Cuál es la mejor cota?

Para $T(n) = (n+1)^2$ podemos decir que $T(n) = O(n^2)$ ó $T(n) = O(n^3)$.

Pero decir que $T(n) = O(n^2)$ es una aseveración *"más fuerte"* del tiempo de ejecución ya que $n^2 < n^3$. En general debemos tomar la *"menor"* de todas las cotas posibles.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Simplificación de expresiones para tiempos de ejecución de los programas

$$T(n) = (3n^3 + 2n^2 + 6)\, n^5 + 2^n + 16n!$$

aplicando la regla de la suma,

$$(3n^3 + 2n^2 + 6) = O(n^3)$$

Aplicando la regla del producto,

$$(3n^3 + 2n^2 + 6)\, n^5 = O(n^8)$$

$$T(n) = \overbrace{(3n^3 + 2n^2 + 6)\, n^5}^{O(n^8)} + 2^n + 16n!$$

Notación asintótica (cont.)

Simplificación de expresiones para tiempos de ejecución de los programas

$$T(n) = \overbrace{\left(3n^3 + 2n^2 + 6\right) n^5}^{O(n^8)} + 2^n + 16n!$$

El término que gobierna es el último de manera que

$$T(n) = O(n!)$$

Muchas veces los diferentes términos que aparecen en la expresión para el tiempo de ejecución corresponde a diferentes partes del programa, de manera que, como ganancia adicional, la notación asintótica nos indica cuáles son las partes del programa que requieren más tiempo de cálculo y, por lo tanto, deben ser eventualmente optimizadas.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Determinación experimental de la tasa de crecimiento

A veces es difícil determinar en forma analítica la tasa de crecimiento del tiempo de ejecución de un algoritmo.

En tales casos puede ser útil determinarla aunque sea en forma *"experimental"* es decir corriendo el programa para una serie de valores de n, tomar los tiempos de ejecución y a partir de estos datos obtener la tasa de crecimiento. Por ejemplo, para el algoritmo heurístico para particionar grafos, si no pudiéramos encontrar el orden de convergencia, entonces ejecutamos el programa con una serie de valores de n obteniendo los valores siguientes

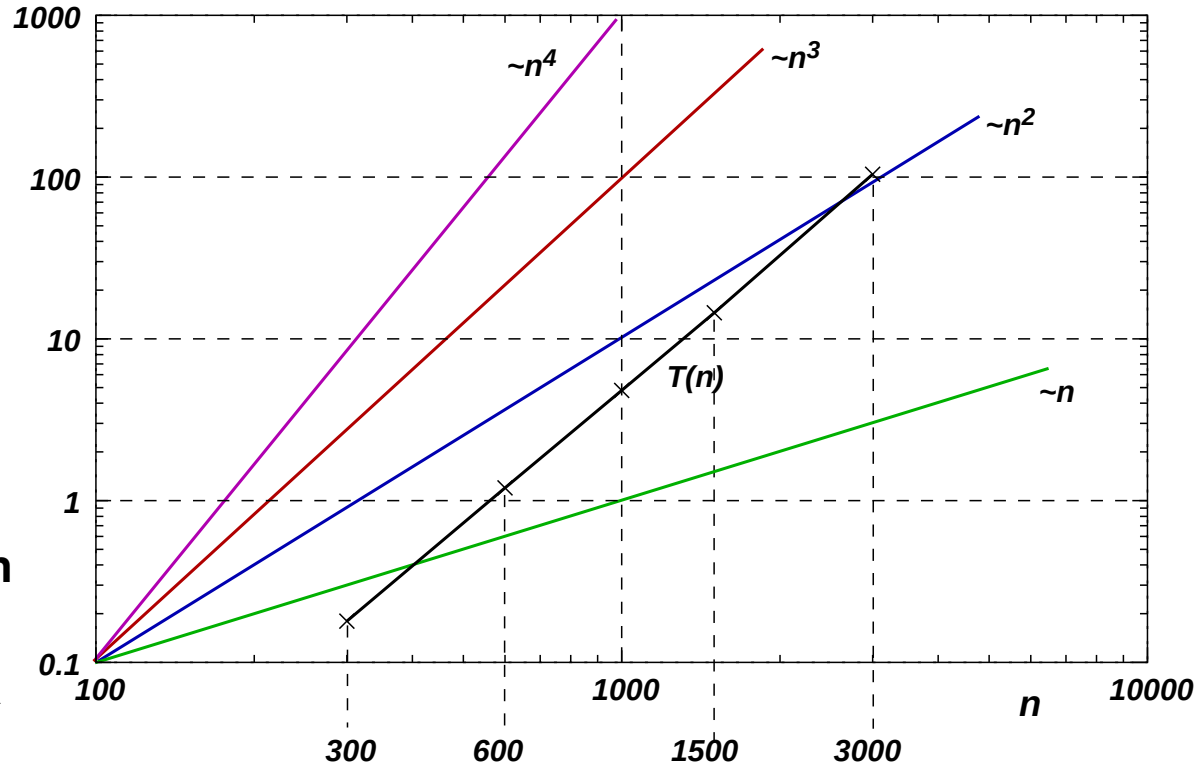
n	$T(n)$ [seg]
300	0.2
600	1.2
1000	4.8
1500	14.5
3000	104.0

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Determinación experimental de la tasa de crecimiento

Graficando los valores en ejes logarítmicos, las funciones de tipo potencia $\propto n^{\alpha}$ resultan ser rectas cuya pendiente es proporcional a α. Además, curvas que difieren en una constante multiplicativa resultan ser simplemente desplazadas según la dirección vertical.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Otros recursos computacionales

Las técnicas desarrolladas en esta sección pueden aplicarse a cualquier otro tipo de recurso computacional, no sólo para el tiempo de ejecución. Otro recurso muy importante es la memoria total requerida. Veremos, por ejemplo, en el capítulo sobre algoritmos de ordenación que el algoritmo de ordenación por *"intercalamiento"* (*merge-sort*) para vectores tiene un tiempo de ejecución $O(n \log n)$ en el caso promedio, pero llega a ser $O(n^2)$ en el peor caso. Pero existe una versión modificada que es siempre $O(n \log n)$, sin embargo la versión modificada requiere una memoria adicional que es en promedio $O(\sqrt{n})$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Algoritmos P y NP

Si bien para ciertos problemas (como el de colorear grafos) no se conocen algoritmos con tiempo de ejecución polinomial, es muy difícil demostrar que realmente es así, es decir que para ese problema no existe ningún algoritmo de complejidad polinomial.

Para ser más precisos hay que introducir una serie de conceptos nuevos. Por empezar, cuando se habla de tiempos de ejecución se refiere a instrucciones realizadas en una *"máquina de Turing"*, que es una abstracción de la computadora más simple posible, con un juego de instrucciones reducido. Una *"máquina de Turing no determinística"* es una máquina de Turing que en cada paso puede *invocar* un cierto número de instrucciones, y no una sola instruccion como es el caso de la máquina de Turing determinística. En cierta forma, es como si una máquina de Turing no-determinística pudiera invocar otras series de máquinas de Turing, de manera que en vez de tener un *"camino de cómputo"*, como es usual en una computadora secuencial, tenemos un *"árbol de cómputo"*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación asintótica (cont.)

Un problema es *"NP"* si tiene un tiempo de ejecución polinomial en una máquina de Turing no determinística (*NP* significa aquí *non-deterministic polynomial*, y no *non-polynomial*). La pregunta del millón de dólares (literalmente!, ver `http://www.claymath.org`) es si existen problemas en NP para los cuales no exista un algoritmo polinomial.

Una forma de simplificar las cosas es demostrar que un problema se *"reduce"* a otro. Por ejemplo, el problema de hallar la mediana de un conjunto de N números (es decir el número tal que existen $N/2$ números menores o iguales que él y otros tantos $N/2$ mayores o iguales) se reduce a poner los objetos en un vector y ordenarlo, ya que una vez ordenado basta con tomar el elemento de la posición media. De manera que si tenemos un cierto algoritmo con una complejidad algorítmica para el ordenamiento, automáticamente tenemos una cota superior para el problema de la mediana.

Notación asintótica (cont.)

Se dice que un problema es *"NP-completo"* (NPC) si cualquier problema de NP se puede reducir a ese problema. Esto quiere decir, que los problemas de NPC son los candidatos a tener la más alta complejidad algorítmica de NP. Se ha demostrado que varios problemas pertenecen a NPC, entre ellos el *Problema del Agente Viajero*. Si se puede demostrar que algún problema de NPC tiene complejidad algorítmica no-polinomial entonces todos los problemas de NPC tendrán la misma complejidad algorítmica. Por otra parte, si se encuentra algún algoritmo de tiempo polinomial para un problema de NPC entonces todos los problemas de NPC (y por lo tanto de NP) serán P. De aquí la famosa forma de poner la pregunta del millón que es: *¿"Es P=NP"?* (ver más en `http://www.wikipedia.org`).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Conteo de operaciones

Comenzaremos por asumir que no hay llamadas recursivas en el programa. La regla básica para calcular el tiempo de ejecución de un programa es ir *desde los lazos o construcciones más internas hacia las más externas*. Se comienza asignando un costo computacional a las sentencias básicas. A partir de esto se puede calcular el costo de un bloque, sumando los tiempos de cada sentencia. Lo mismo se aplica para funciones y otras construcciones sintácticas que iremos analizando a continuación.

Bloques if

Para evaluar el tiempo de un bloque *if*

```
if(<cond>) {
   <body>
}
```

podemos o bien considerar el peor caso, asumiendo que *<body>* se ejecuta siempre

$$T_{\text{peor}} = T_{\text{cond}} + T_{\text{body}}$$

o, en el caso promedio, calcular la probabilidad P de que *<cond>* de verdadero. En ese caso

$$T_{\text{prom}} = T_{\text{cond}} + PT_{\text{body}}$$

Notar que T_{cond} no está afectado por P ya que la condición se evalúa siempre.

Bloques if (cont.)

En el caso de que tenga un bloque `else`, entonces

```
if(<cond>) {
  <body-true>
} else {
  <body-false>
}
```

podemos considerar,

$$
\begin{aligned}
T_{\text{peor}} &= T_{\text{cond}} + \max(T_{\text{body-true}}, T_{\text{body-false}}) \\
&\leq T_{\text{cond}} + T_{\text{body-true}} + T_{\text{body-false}} \\
T_{\text{prom}} &= T_{\text{cond}} + P\,T_{\text{body-true}} + (1-P)\,T_{\text{body-false}}
\end{aligned}
$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Lazos

El caso más simple es cuando el lazo se ejecuta un número fijo de veces, y el cuerpo del lazo tiene un tiempo de ejecución constante,

```
for (i=0; i<N; i++) {
    <body>
}
```

$$T = T_{\text{ini}} + N(T_{\text{body}} + T_{\text{inc}} + T_{\text{stop}})$$

- T_{ini} **es el tiempo de ejecución de la parte de** ***"inicialización"*** **del lazo, en este caso** `i=0`**,**
- T_{inc} **es el tiempo de ejecución de la parte de** ***"incremento"*** **del contador del lazo, en este caso,** `i++` **y**
- T_{stop} **es el tiempo de ejecución de la parte de** ***"detención"*** **del contador del lazo, en este caso,** `i<N`**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Lazos (cont.)

En el caso más general, cuando T_{body} no es constante, debemos evaluar explícitamente la suma de todas las contribuciones,

$$T = T_{\text{ini}} + \sum_{i=0}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}).$$

Algunas veces es difícil calcular una expresión analítica para tales sumas. Si podemos determinar una cierta tasa de crecimiento para todos los términos

$$T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(f(n)), \quad \textbf{para todo } i$$

entonces,

$$T \leq T_{\text{ini}} + N \max_{i=1}^{N-1}(T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}) = T_{\text{ini}} + O(Nf(n))$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

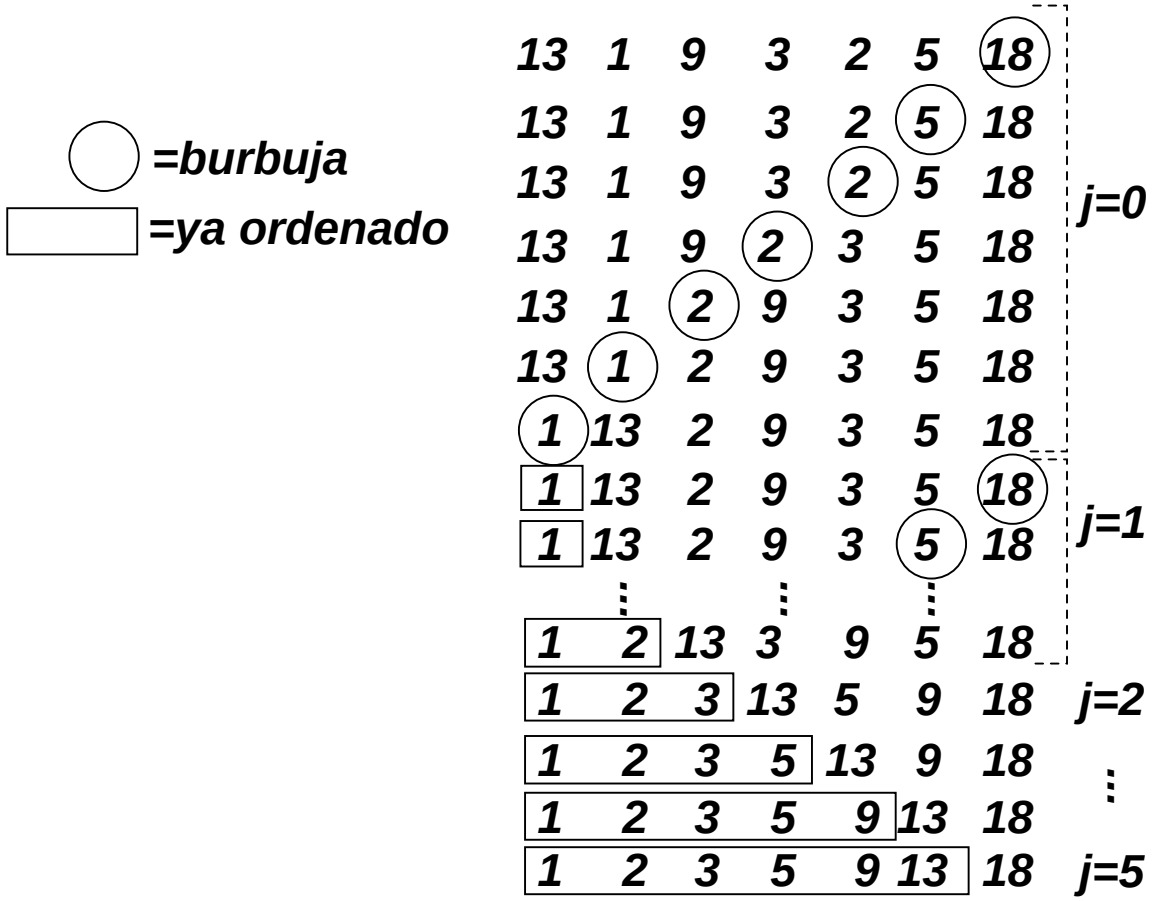
Lazos (cont.)

Más difícil aún es el caso en que el número de veces que se ejecuta el lazo no se conoce a priori, por ejemplo un lazo `while` como el siguiente

```
while (<cond>) {
    <body>
}
```

En este caso debemos determinar también el número de veces que se ejecutará el lazo.

Ej: Bubble-sort



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

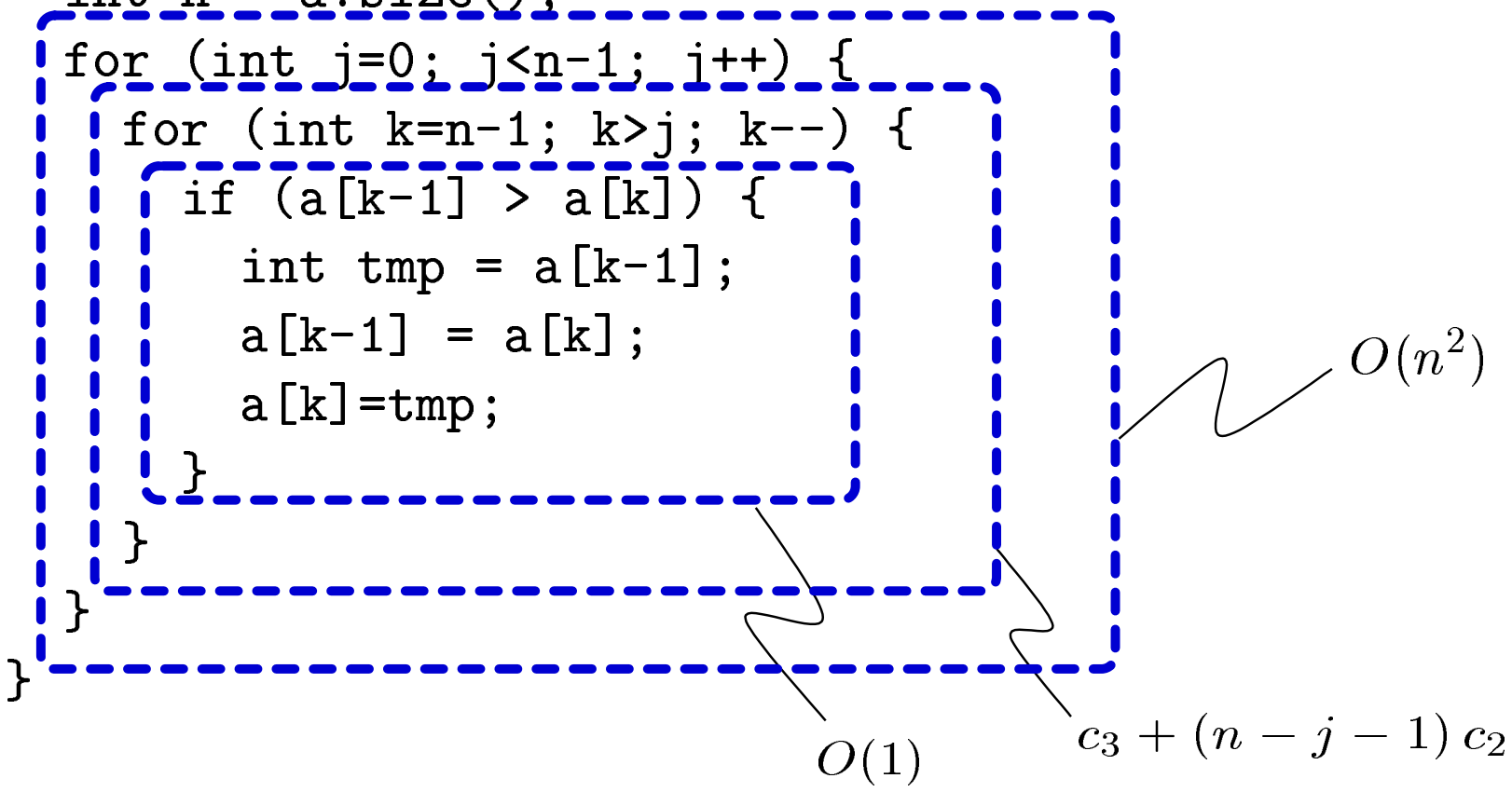
Ej: Bubble-sort (cont.)

- `bubble_sort(a)` ordena de menor a mayor `a`.
- Para `j=0` el menor elemento de todos es insertado en `a[0]` mediante una serie de intercambios.
- A partir de ahí `a[0]` no es tocado más.
- Para `j=1` el mismo procedimiento es aplicado al rango de índices que va desde `j=1` hasta `j=n-1`, hasta ordenar todo el vector.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ej: Bubble-sort (cont.)

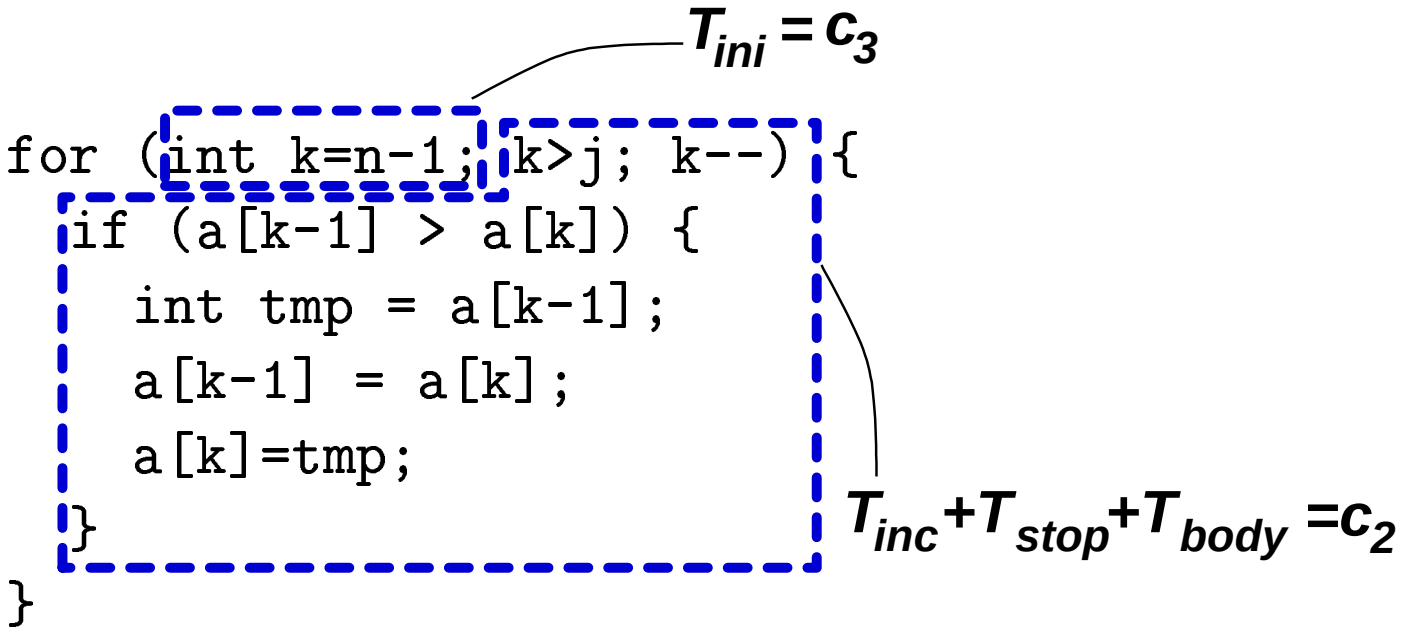
```
void bubble_sort(vector<int> &a) {
  int n = a.size();
  for (int j=0; j<n-1; j++) {
    for (int k=n-1; k>j; k--) {
      if (a[k-1] > a[k]) {
        int tmp = a[k-1];
        a[k-1] = a[k];
        a[k]=tmp;
      }
    }
  }
}
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ej: Bubble-sort (cont.)

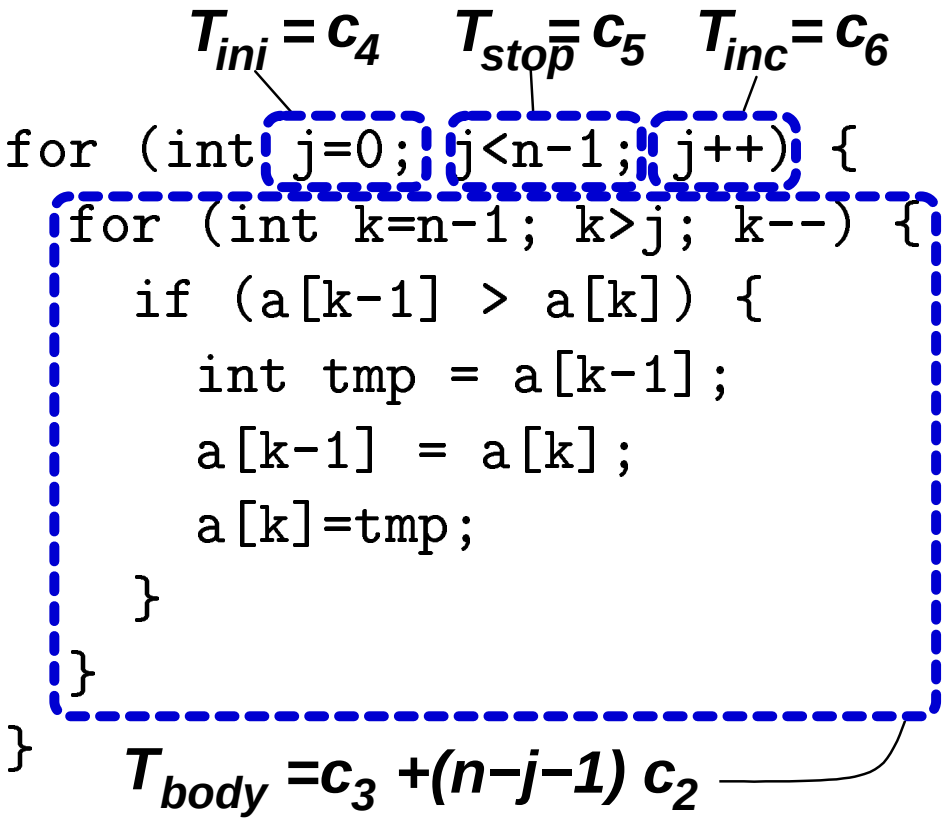
Lazo sobre k, se ejecuta $n-j-1$ veces:



$$T = c_3 + (n - j - 1)\, c_2$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Lazo sobre j, se ejecuta $n-1$ veces:



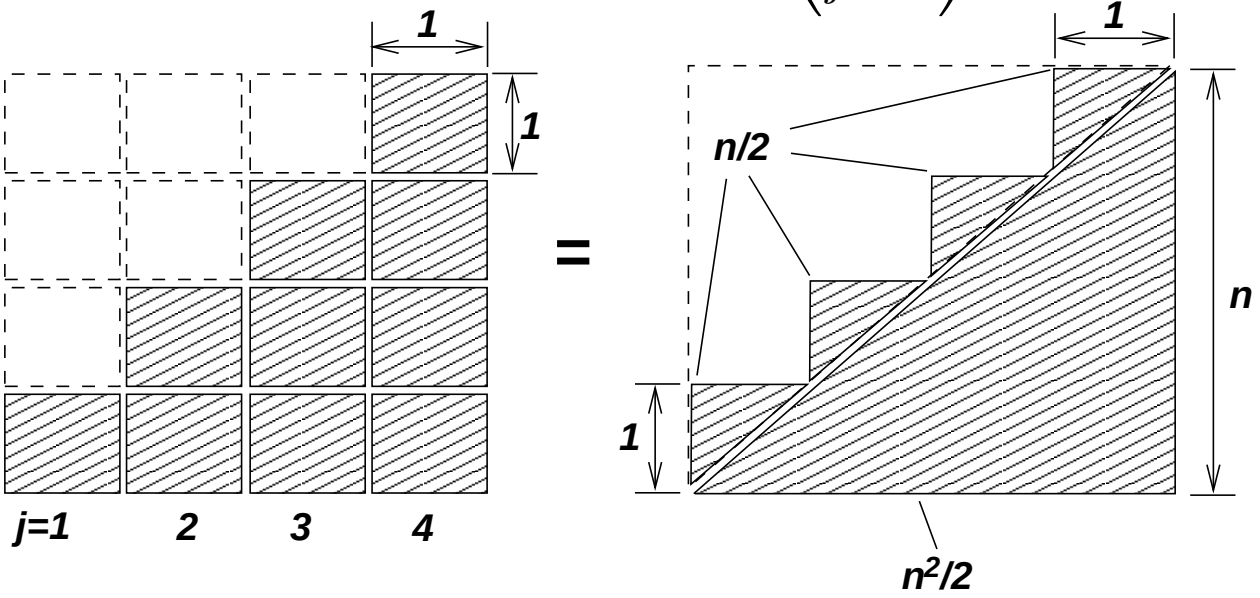
$$T = c_4 + (n-1)(c_3 + c_5 + c_6) + c_2 \sum_{j=0}^{n-2} (n-j-1)$$

Ej: Bubble-sort (cont.)

$$\sum_{j=0}^{n-2}(n-j-1) = (n-1)+(n-2)+\ldots+1 = \left(\sum_{j=1}^{n} j\right) - n.$$

Consideremos:

$$\sum_{j=1}^{n} j =$$



$$= \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ej: Bubble-sort (cont.)

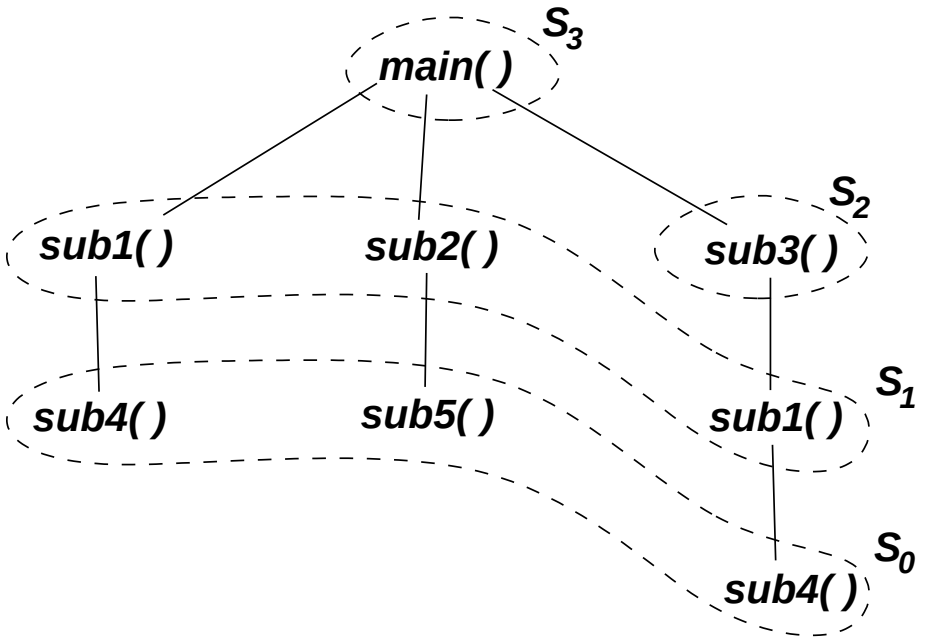
$$T(n) = \overbrace{c_4}^{O(1)} + \overbrace{(n-1)(c_3+c_5+c_6)}^{O(n)} + \overbrace{c_2\,\frac{n(n-1)}{2}}^{O(n^2)}$$

$$= O(\frac{n^2}{2})$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

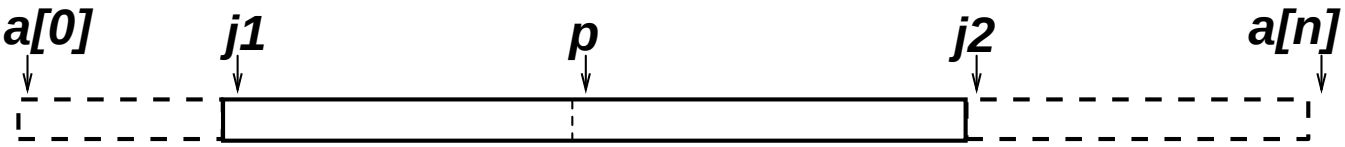
Llamadas a rutinas

- **Calculados los tiempos de rutinas que no llaman a otras rutinas (S_0), calculamos el de aquellas rutinas que *sólo* llaman a rutinas de S_0 (llamémoslas S_1).**
- **Se asigna a las líneas con llamadas a rutinas de S_0 de acuerdo con el tiempo de ejecución previamente calculado, como si fuera una instrucción más del lenguaje. Por ejemplo podemos usar a `bubble` como una instrucción cuyo costo es $O(n^2)$.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Llamadas recursivas




```
1  int bsearch2(vector<int> &a,int k,int j1, int j2) {
2    if (j1==j2-1) {
3      if (a[j1]==k) return j1;
4      else return j2;
5    } else {
6      int p = (j1+j2)/2;
7      if (k<a[p]) return bsearch2(a,k,j1,p);
8      else return bsearch2(a,k,p,j2);
9    }
10 }
11
12 int bsearch(vector<int> &a,int k) {
13   int n = a.size();
14   if (k<a[0]) return 0;
15   else return bsearch2(a,k,0,n);
16 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Llamadas recursivas (cont.)

$$T(m) = \begin{cases} c & ; \textbf{ si } m = 1; \\ d + T(m/2) & ; \textbf{ si } m > 1; \end{cases}$$

$$T(2) = d + T(1) = d + c$$

$$T(4) = d + T(2) = 2d + c$$

$$T(8) = d + T(4) = 3d + c$$

$$\vdots$$

$$T(2^p) = d + T(2^{p-1}) = pd + c$$

como $p = \log_2 m$**, vemos que**

$$T(m) = d \log_2 m + c = O(\log_2 m)$$

Llamadas recursivas (cont.)

Hemos acotado el tiempo del `if` usando `max`. Si tomamos la suma entonces llegamos a una estimación,

$$T(m) = \begin{cases} c & ; \textbf{ si } m = 1; \\ d + 2T(m/2) & ; \textbf{ si } m > 1; \end{cases}$$

$$T(2) = d + 2T(1) = d + 2c$$

$$T(4) = d + 2T(2) = d + 2d + 4c = 3d + 4c$$

$$T(8) = d + 2T(4) = d + 6d + 8c = 7d + 8c$$

$$\vdots$$

$$T(n) = (n-1)d + nc \leq (c+d)n = O(n), \quad n = 2^p$$

Llamadas recursivas (cont.)

Notar entonces que si bien las dos cotas (por el máximo y por la suma) son válidas, la del máximo da una mejor estimación (más baja) en este caso.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tipos de datos abstractos fundamentales

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

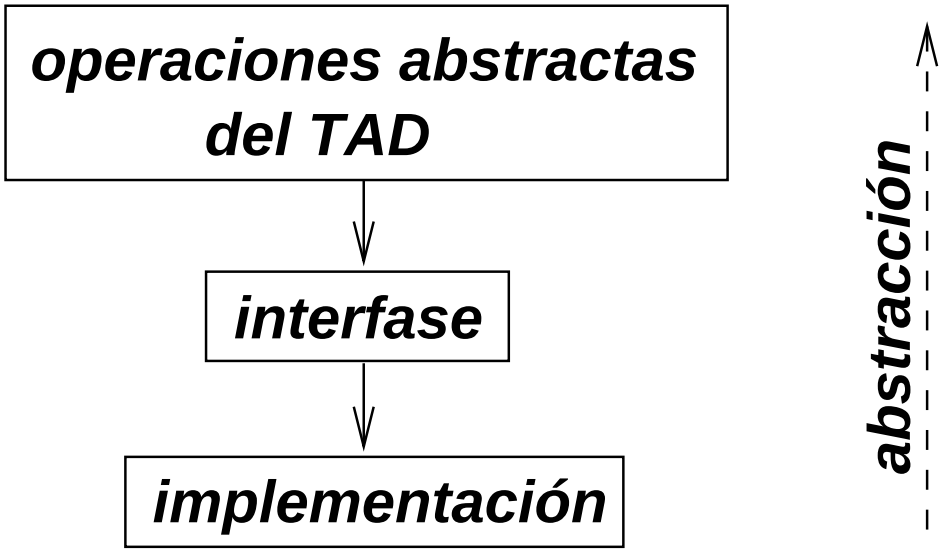
Ventajas del uso de estructuras de datos

- **Se ahorra tiempo de programación ya que no es necesario codificar.**
- **Estas implementaciones suelen ser eficientes y robustas.**
- **Se separan dos capas de código bien diferentes, por una parte el algoritmo que escribe el programador, y por otro las rutinas de acceso a las diferentes estructuras.**
- **Existen estimaciones bastante uniformes de los tiempos de ejecución de las diferentes operaciones.**
- **Las funciones asociadas a cada estructura son relativamente independientes del lenguaje o la implementación en particular. Así, una vez que se plantea un algoritmo en términos de operaciones sobre una estructura es fácil implementarlo en una variedad de lenguajes con una performance similar.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Niveles de abstracción en la definición de un TAD

TAD = ***"Tipo Abstracto de Datos"***



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Op. abstractas y características del TAD conjunto

- **Contiene elementos, los cuales deben ser diferentes entre sí.**
- **No existe un orden particular entre los elementos del conjunto.**
- **Se pueden insertar o eliminar elementos del mismo.**
- **Dado un elemento se puede preguntar si está dentro del conjunto o no.**
- **Se pueden hacer las operaciones binarias bien conocidas entre conjuntos a saber, unión, intersección y diferencia.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase STL del TAD CONJUNTO

```
1  template<class T>
2  class set {
3  public:
4    class iterator { /* . . . */ };
5    void insert(T x);
6    void erase(iterator p);
7    void erase(T x);
8    iterator find(T x);
9    iterator begin();
10   iterator end();
11 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase

- `s.insert(x)` inserta un elemento en el conjunto. Si el elemento ya estaba en el conjunto `s` queda inalterado.
- `p=s.find(x)` devuelve el iterador para el elemento `x`. Si `x` no está en `s` entonces devuelve un iterador especial `end()`. En consecuencia, la expresión lógica para saber si un elemento está en `s` es

```
1 if(s.find(x)==s.end()) {
2   // 'x' no esta en 's'
3   // . . .
4 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- `s.erase(p)` elimina el elemento que está en el iterador `p` en `s`.
  `s.erase(x)` elimina el elemento `x` (si está en el conjunto).
- La unión de dos conjuntos, por ejemplo $C = A \cup B$ podemos lograrla insertando los elementos de A y B en C:

```
1 set A,B,C;
2 // Pone elementos en A y B
3 // . . .
4 C.insert(A.begin(),A.end());
5 C.insert(B.begin(),B.end());
```

Normalmente en C/C++ la interfase está definida en los headers de las respectivas clases.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- **Todas las operaciones binarias con conjuntos se pueden realizar con algoritmos genéricos definidos en el header `algorithm`, usando el adaptador `inserter`. Típicamente:**

  ▷ $C = A \cup B$

```
set_union(a.begin(),a.end(),b.begin(),b.end(),
          inserter(c,c.begin()));
```

  ▷ $C = A - B$

```
set_difference(a.begin(),a.end(),b.begin(),b.end(),
               inserter(c,c.begin()));
```

  ▷ $C = A \cap B$

```
set_intersection(a.begin(),a.end(),b.begin(),b.end(),
                 inserter(c,c.begin()));
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tipos de datos abstractos fundamentales

En este capítulo se estudiarán varios tipos de datos básicos. Para cada uno de estos TAD se discutirán en el siguiente orden

1. **Sus operaciones abstractas.**
2. **Una interfase en C++ y ejemplos de uso con esta interfase.**
3. **Una o más implementaciones de esa interfase, discutiendo las ventajas y desventajas de cada una, tiempos de ejecución ...**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tipos de datos abstractos fundamentales (cont.)

Para los primeros TAD que estudiaremos, veremos dos interfases,

- **Una interfase básica, (sin templates, sobrecarga de operadores, clases anidadas...)**
- **Una interfase más avanzada, compatible con las STL, usando templates, sobrecarga de operadores y clases anidadas y ejemplos de uso de esta interfase.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD lista

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD lista. Descripción matemática

- **Una lista es una secuencia de cero o más elementos de un tipo determinado `elem_t` (por ejemplo `int` o `double`).**
- **Representación impresa ($n \geq 0$ es el número de elementos)**

$$L = (a_0, a_1, \ldots, a_{n-1}) \tag{15}$$

- **Cada a_i es de tipo `elem_t`.**
- **a_0 es el primer elemento y a_{n-1} es el último elemento de la lista.**
- **Si $n = 0$ decimos que la lista *"está vacía"*.**
- **n es la *"longitud"* de la lista.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas. Descripción matemática

- **Existe un orden lineal entre los elementos de la lista.**
- **Por ejemplo, las siguientes listas son distintas**

$$(1, 3, 7, 4) \neq (3, 7, 4, 1)$$

- **Decimos que a_i *"está en la posición entera i"*.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas. Descripción matemática

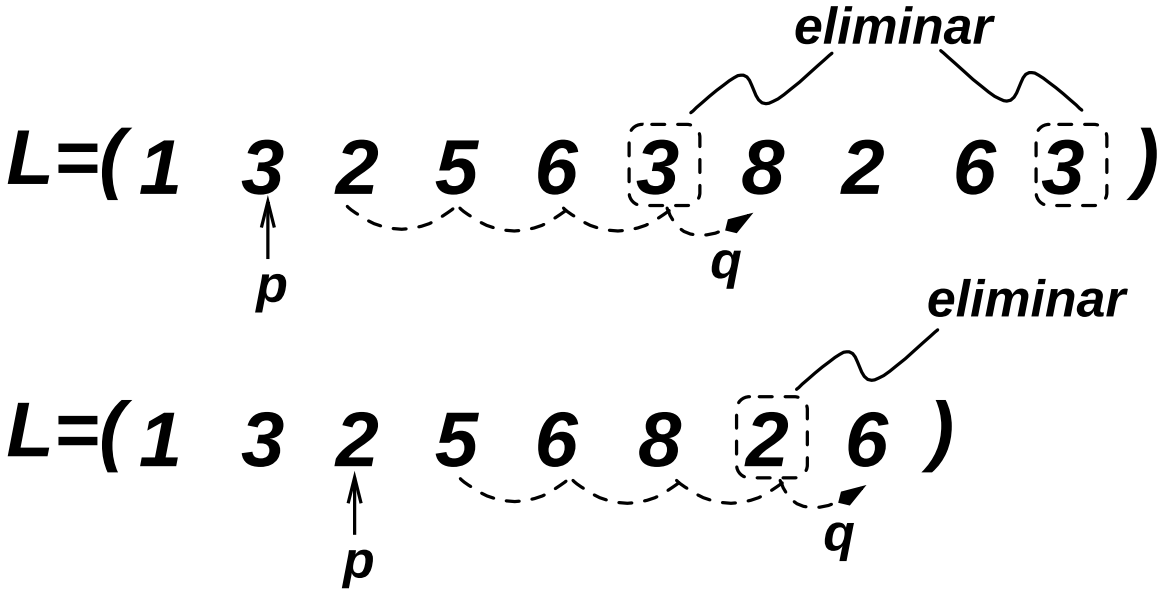
- **Introducimos una posición ficticia n que *está fuera de la lista*.** $(1, 3, 7, 4, < \text{end} >)$
- **Posiciones *"dereferenciables"*: $0 \leq i \leq n-1$ (Podemos dereferenciar la posición y obtener el elemento correspondiente).**
- **Por analogía con el puntero nulo:**

```
1 int j = 23;
2 int *p = &j; // p es 'derefenciable'
3 cout << "el valor apuntado por p es: "
4       << *p << endl; // OK (imprime 23)
5
6 int *q = NULL; // Atencion! p no es 'dereferenciable'
7 cout << "el valor apuntado por q es: "
8       << *q << endl; // ERROR
```

- **Como n va variando convendrá tener `end()` que retorne la posición ficticia.**

Operaciones abstractas sobre listas. Algoritmo *purge*

Consigna: **Eliminar elementos repetidos de una lista.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones abstractas sobre listas. (cont.)

Este problema sugiere las siguientes operaciones abstractas

- **Dada una posición i, *"insertar"* el elemento x en esa posición, por ejemplo**

$$L = (1, 3, 7)$$

inserta 5 en la posición 2 (16)

$$\rightarrow L = (1, 3, 5, 7)$$

Notar que el elemento 7, que estaba en la posición 2, se desplaza hacia el fondo, y termina en la posición 3. Notar que es válido insertar en cualquier posición dereferenciable, o en la posición ficticia `end()`

Operaciones abstractas sobre listas. (cont.)

- **Dada una posición i, *"suprimir"* el elemento que se encuentra en la misma. Por ejemplo,**

$$L = (1, 3, 5, 7)$$

suprime elemento en la posición 2 (17)

$$\rightarrow L = (1, 3, 7)$$

Notar que esta operación es, en cierta forma, la inversa de insertar. Si, como en el ejemplo anterior, insertamos un elemento en la posición i y después suprimimos en esa misma posición, entonces la lista queda inalterada. Notar que *sólo* es válido suprimir en las posiciones dereferenciables.

Posiciones abstractas

Si representáramos las posiciones como enteros, entonces avanzar la posición podría efectuarse con la sencilla operación de enteros $i \leftarrow i + 1$, pero es deseable pensar en las posiciones como entidades abstractas, no necesariamente enteros y por lo tanto para las cuales no necesariamente es válido hacer operaciones de enteros. Esto lo sugiere la experiencia previa de cursos básicos de programación donde se ha visto que las listas se representan por celdas encadenadas por punteros. En estos casos, puede ser deseable representar a las posiciones como punteros a las celdas. De manera que asumiremos que las posiciones son objetos abstractos.

Manejo de posiciones abstractas

- **Acceder al elemento en la posición p, tanto para modificar el valor ($a_p \leftarrow x$) como para acceder al valor ($x \leftarrow a_p$).**
- **Avanzar una posición, es decir, dada una posición p correspondiente al elemento a_i, retornar la posición q correspondiente al elemento a_{i+1}. (Como mencionamos previamente, no es necesariamente $q = p + 1$, o más aún, pueden no estar definidas estas operaciones aritméticas sobre las posiciones p y q.)**
- **Retornar la primera posición de la lista, es decir, la correspondiente al elemento a_0.**
- **Retornar la posición ficticia al final de la lista, es decir, la correspondiente a n.**

Interfase para listas

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una interfase simple para listas

```
1  class iterator_t { /* . . . */ };
2
3  class list {
4  private:
5    // . . .
6  public:
7    // . . .
8    iterator_t insert(iterator_t p,elem_t x);
9    iterator_t erase(iterator_t p);
10   elem_t & retrieve(iterator_t p);
11   iterator_t next(iterator_t p);
12   iterator_t begin();
13   iterator_t end();
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una interfase simple para listas (cont.)

- `q = insert(p,x)`: inserta el elemento `x` en la posición `p`, devolviendo una posición `q` al elemento insertado. Todas las posiciones de `p` en adelante (incluyendo `p`) pasan a ser inválidas, por eso la función devuelve a `q`, la nueva posición insertada, ya que la anterior `p` es inválida. Es válido insertar en cualquier posición dereferenciable o no dereferenciable, es decir, que es válido insertar también en la posición ficticia.
  Ejemplo: L=$(1,3,7,4)$, `p` *"apunta"* al 3.

```
1  L.insert(p,5); // queda L=(1,5,3,7,4)
2  x = L.retrieve(p); // ERRROR! p es invalido
```

Lo correcto es

```
1  p = L.insert(p,5); // queda L=(1,5,3,7,4)
2  x = L.retrieve(p); // OK! hace x=5
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una interfase simple para listas (cont.)

- `q = erase(p)`: elimina el elemento en la posición `p`, devolviendo una posición `q` al elemento que previamente estaba en la posición siguiente a `p`. Todas las posiciones de `p` en adelante (incluyendo `p`) pasan a ser inválidas. Sólo es válido suprimir en las posiciones dereferenciables de la lista.

  Ejemplo: L=$(1, 3, 7, 4)$, `p` *"apunta"* al 3.

```
1  L.erase(p); // queda L=(1,7,4)
2  x = L.retrieve(p); // ERRROR! p es invalido
```

  Lo correcto es

```
1  p = L.erase(p); // queda L=(1,7,4), p apunta a 7
2  x = L.retrieve(p); // OK! hace x=7
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- `x=retrieve(p)`: *"recupera"* el elemento en la posición `p`, devolviendo una referencia al mismo, de manera que es válido hacer tanto `x = L.retrieve(p)` como `L.retrieve(p)=x`. Se puede aplicar a cualquier posición `p` dereferenciable. Usado en el miembro derecho no modifica a la lista, en el miembro izquierdo sí. Notar que retorna una *referencia* al elemento correspondiente de manera que éste puede ser cambiado, es decir, puede ser usado como un *"valor asignable"* (*"left hand side value"*).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- `q = next(p)`: dada una posición dereferenciable `p`, devuelve la posición del siguiente elemento. Si `p` es la última posición dereferenciable, entonces devuelve la posición ficticia. No modifica la lista.
- `begin`: devuelve la posición del primer elemento de la lista.
- `end`: devuelve la posición ficticia (no dereferenciable), después del final de la lista.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- **Posiciones inválidas: Un elemento a tener en cuenta es que las funciones que modifican la lista como `insert` and `erase`, convierten en inválidas algunas de las posiciones de la lista, normalmente desde el punto de inserción/supresión en adelante, incluyendo `end()`.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

Por ejemplo,

```
1  iterator_t p,q,r;
2  list L;
3  elem_t x,y,z;
4  //...
5  // p es una posicion dereferenciable
6  q = L.next(p);
7  r = L.end();
8  L.erase(p);
9  x = L.retrieve(p); // incorrecto
10 y = L.retrieve(q); // incorrecto
11 L.insert(r,z); // incorrecto
```

ya que `p,q,r` ya no son válidos (están después de la posición borrada `p`).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

La forma correcta de escribir el código anterior es

```
1  iterator_t p,q,r;
2  list L;
3  elem_t x,y,z;
4  //. . .
5  // p es una posicion dereferenciable
6  p = L.erase(p);
7  x = L.retrieve(p); // correcto
8  q = L.next(p);
9  y = L.retrieve(q); // correcto
10 r = L.end();
11 L.insert(r,z); // correcto
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- **Las posiciones sólo se acceden a través de funciones de la clase: Las únicas operaciones válidas con posiciones son**
  - ▷ **Asignar:**

```
p = L.begin();
q = L.end();
```

  - ▷ **Avanzar:**

```
q = L.next(p);
```

  - ▷ **Acceder al elemento:**

```
x = L.retrieve(p);
L.retrieve(q) = y;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la interfase *(cont....)*

- **Copiar:**

```
q = p;
```

- **Comparar: Notar que sólo se puede comparar por igualdad o desigualdad, no por operadores de comparación, como `<` ó `>`.**

```
q == p
r != L.end();
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones que retornan referencias

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones que retornan referencias

Consigna: **Escribir una función que duplique el valor mínimo de un arreglo de enteros.**

```
1 int min(int *v,int n,int *jmin);
2 . . .
3
4 int jmin;
5 int m = min(v,n,&jmin);
6 v[jmin] = 2*v[jmin];
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones que retornan referencias (cont.)

Usando punteros:

```
1  int *min(int *v,int n) {
2    int x = v[0];
3    int jmin = 0;
4    for (int k=1; k<n; k++) {
5      if (v[k]<x) {
6        jmin = k;
7        x = v[jmin];
8      }
9    }
10   return &v[jmin];
11 }
12
13 void print(int *v,int n) {
14   cout << "Vector: (";
15   for (int j=0; j<n; j++) cout << v[j] << " ";
16   cout << "), valor minimo: " << *min(v,n) << endl;
17 }
18
19 int main() {
20   int v[] = {6,5,1,4,2,3};
21   int n = 6;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22
23    print(v,n);
24    for (int j=0; j<6; j++) {
25      *min(v,n) = 2* (*min(v,n));
26      print(v,n);
27    }
28  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones que retornan referencias (cont.)

Salida:

```
[mstorti@spider aedsrc]$ ptrexa
Vector: (6 5 1 4 2 3 ), valor minimo: 1
Vector: (6 5 2 4 2 3 ), valor minimo: 2
Vector: (6 5 4 4 2 3 ), valor minimo: 2
Vector: (6 5 4 4 4 3 ), valor minimo: 3
Vector: (6 5 4 4 4 6 ), valor minimo: 4
Vector: (6 5 8 4 4 6 ), valor minimo: 4
Vector: (6 5 8 8 4 6 ), valor minimo: 4
[mstorti@spider aedsrc]$
```

Funciones que retornan referencias (cont.)

Usando referencias:

```
1  int &min(int *v,int n) {
2    int x = v[0];
3    int jmin = 0;
4    for (int k=1; k<n; k++) {
5      if (v[k]<x) {
6        jmin = k;
7        x = v[jmin];
8      }
9    }
10   return v[jmin];
11 }
12
13 void print(int *v,int n) {
14   cout << "Vector: (";
15   for (int j=0; j<n; j++) cout << v[j] << " ";
16   cout << "), valor minimo: " << min(v,n) << endl;
17 }
18
19 int main() {
20   int v[] = {6,5,1,4,2,3};
21   int n = 6;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22
23    print(v,n);
24    for (int j=0; j<6; j++) {
25      min(v,n) = 2*min(v,n);
26      print(v,n);
27    }
28  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Lista. Ejemplos de uso de la interfase

```
1  void purge(list &L) {
2    iterator_t p,q;
3    p = L.begin();
4    while (p!=L.end()) {
5      q = L.next(p);
6      while (q!=L.end()) {
7        if (L.retrieve(p)==L.retrieve(q)) {
8          q = L.erase(q);
9        } else {
10         q = L.next(q);
11       }
12     }
13     p = L.next(p);
14   }
15 }
16
17 int main() {
18   list L;
19   const int M=10;
20   for (int j=0; j<2*M; j++)
21     L.insert(L.end(),irand(M));
22   cout << "Lista antes de purgar: " << endl;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23   print(L);
24   cout << "Purga lista. . . " << endl;
25   purge(L);
26   cout << "Lista despues de purgar: " << endl;
27   print(L);
28 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Lista. Ejemplos de uso de la interfase (cont.)

```
[mstorti@minerva aedsrc]$ purge
Lista antes de purgar:
8 3 7 7 9 1 3 7 2 5 4 6 3 5 9 9 6 7 1 6
Purga lista...
Lista despues de purgar:
8 3 7 9 1 2 5 4 6
[mstorti@minerva aedsrc]$
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

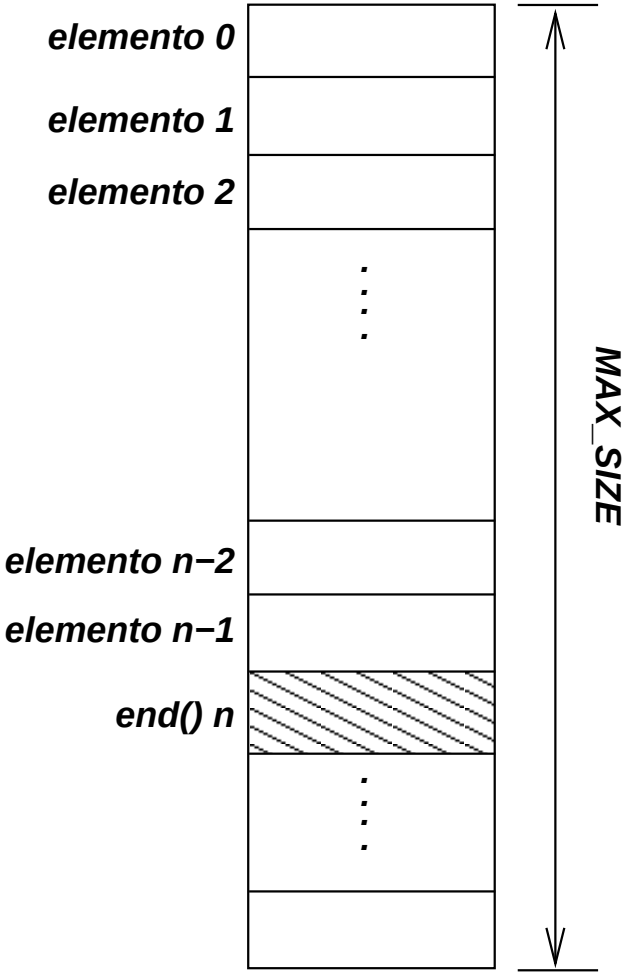
Implementaciones de lista

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas por arreglos

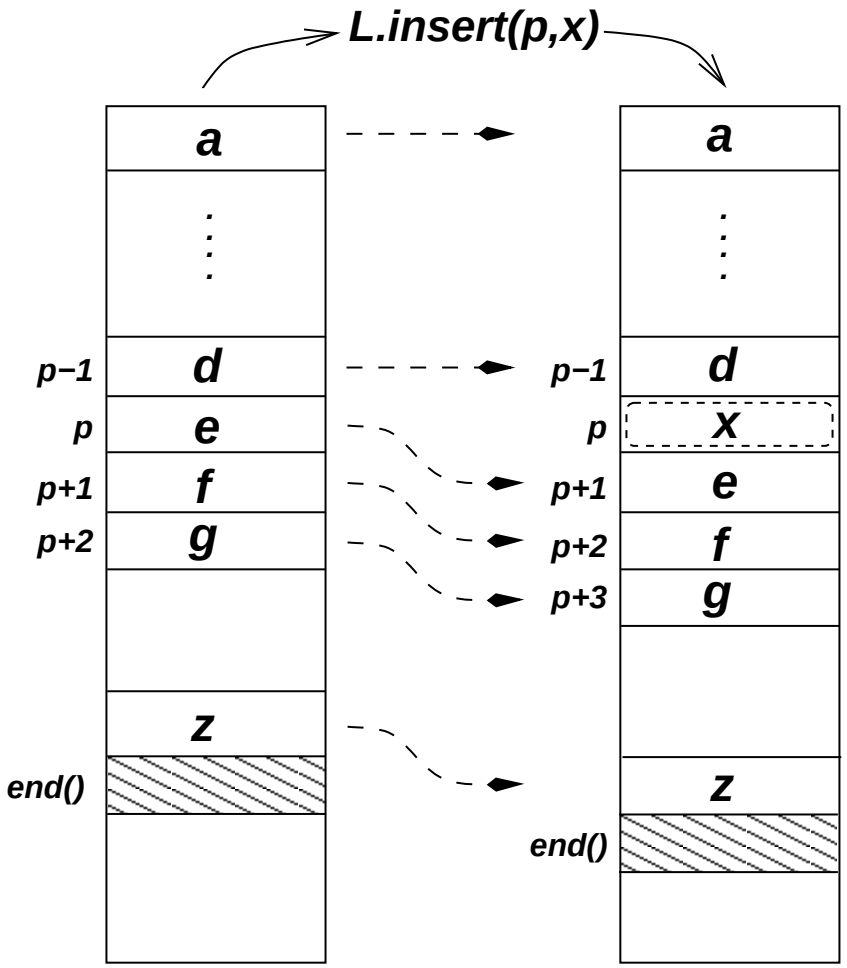
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1   typedef int iterator_t;
2
3   class list {
4   private:
5     static int MAX_SIZE;
6     elem_t *elems;
7     int size;
8   public:
9     list();
10    ~list();
11    iterator_t insert(iterator_t p,elem_t j);
12    iterator_t erase(iterator_t p);
13    iterator_t erase(iterator_t p,iterator_t q);
14    void clear();
15    iterator_t begin();
16    iterator_t end();
17    void print();
18    iterator_t next(iterator_t p);
19    iterator_t prev(iterator_t p);
20    elem_t & retrieve(iterator_t p);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
21      };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1  #include <iostream>
2  #include <aedsrc/lista.h>
3  #include <cstdlib>
4
5  using namespace std;
6  using namespace aed;
7
8  int list::MAX_SIZE=100;
9
10 list::list() : elems(new elem_t[MAX_SIZE]),
11              size(0) { }
12
13 list::~list() { delete[] elems; }
14
15 elem_t &list::retrieve(iterator_t p) {
16   if (p<0 || p>=size) {
17     cout << "p: mala posicion.\n";
18     abort();
19   }
20   return elems[p];
21 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1  iterator_t list::begin() { return 0; }
2
3  iterator_t list::end() { return size; }
4
5  iterator_t list::next(iterator_t p) {
6    if (p<0 || p>=size) {
7      cout << "p: mala posicion.\n";
8      abort();
9    }
10   return p+1;
11 }
12
13 iterator_t list::prev(iterator_t p) {
14   if (p<=0 || p>size) {
15     cout << "p: mala posicion.\n";
16     abort();
17   }
18   return p-1;
19 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1  iterator_t list::insert(iterator_t p,elem_t k) {
2    if (size>=MAX_SIZE) {
3      cout << "La lista esta llena.\n";
4      abort();
5    }
6    if (p<0 || p>size) {
7      cout << "Insertando en posicion invalida.\n";
8      abort();
9    }
10   for (int j=size; j>p; j--) elems[j] = elems[j-1];
11   elems[p] = k;
12   size++;
13   return p;
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1 iterator_t list::erase(iterator_t p) {
2   if (p<0 || p>=size) {
3     cout << "p: posicion invalida.\n";
4     abort();
5   }
6   for (int j=p; j<size-1; j++) elems[j] = elems[j+1];
7   size--;
8   return p;
9 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1  iterator_t list::erase(iterator_t p,iterator_t q) {
2    if (p<0 || p>=size) {
3      cout << "p: posicion invalida.\n";
4      abort();
5    }
6    if (q<0 || q>=size) {
7      cout << "q: posicion invalida.\n";
8      abort();
9    }
10   if (p>q) {
11     cout << "p debe estar antes de q\n";
12     abort();
13   }
14   if (p==q) return p;
15   int shift = q-p;
16   for (int j=p; j<size-shift; j++)
17     elems[j] = elems[j+shift];
18   size -= shift;
19   return p;
20 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas por arreglos (cont.)

```
1  void list::clear() { erase(begin(),end()); }
2
3  void list::print() {
4    iterator_t p = begin();
5    while (p!=end()) {
6      cout << retrieve(p) << " ";
7      p = next(p);
8    }
9    cout << endl;
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Desventajas de la representación por arreglos

- **Almacenamiento rígido**
- **Tiempo de ejecución de `insert(p,x)` y `erase(p)` son $O(n)$ en promedio.**

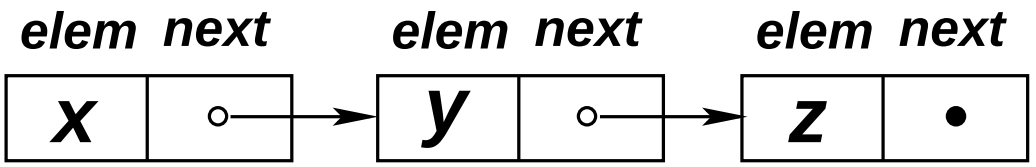
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas por celdas enlazadas

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de l. por celdas enlazadas

```
1  class cell {
2    friend class list;
3    elem_t elem;
4    cell *next;
5    cell() : next(NULL) {}
6  };
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de l. por celdas enlazadas (cont.)

```
1   class cell;
2   typedef cell *iterator_t;
3
4   class list {
5   private:
6     cell *first, *last;
7   public:
8     list();
9     ~list();
10    iterator_t insert(iterator_t p,elem_t j);
11    iterator_t erase(iterator_t p);
12    iterator_t erase(iterator_t p,iterator_t q);
13    void clear();
14    iterator_t begin();
15    iterator_t end();
16    void print();
17    void printd();
18    iterator_t next(iterator_t p);
19    iterator_t prev(iterator_t p);
20    elem_t & retrieve(iterator_t p);
```

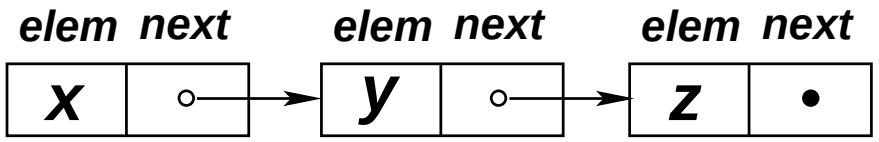

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
21      int size();
22    };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de l. por celdas enlazadas (cont.)

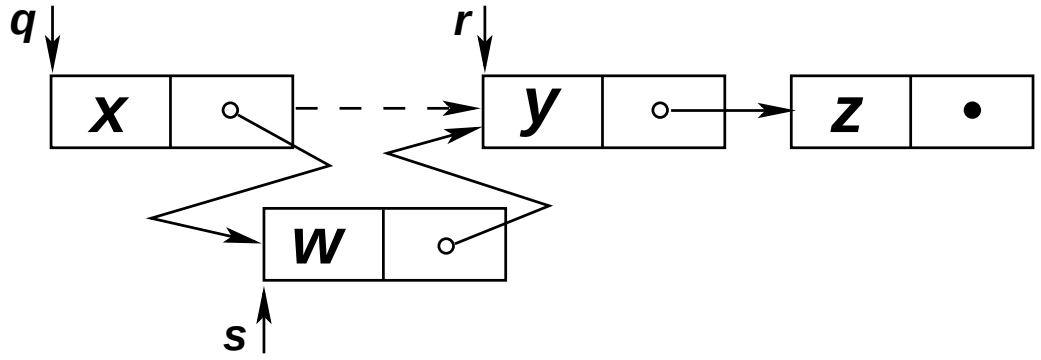
- **Dado un puntero a una celda se puede seguir la cadena en la dirección de los links.**
- **La lista se termina cuando se encuentra el *"terminador"*, es decir un `NULL` en el campo `next`. Está garantizado que `NULL` es un puntero inválido.**
- **No se puede recorrer la lista en el sentido contrario. Dado un puntero a la celda que contiene a `z` no se puede obtener la que contiene a `y`.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El tipo posición

- ¿Cual será `iterator_t`? La elección natural parece ser `cell *`
- Podemos definir como posición el puntero a la celda que contiene al elemento. Ej: El elemento `y` está en la posición `r` y el elemento `x` está en la posición `q`. Recordar `x` e `y` son elementos (tipo `elem_t`), `q` y `r` son de tipo `cell *`.

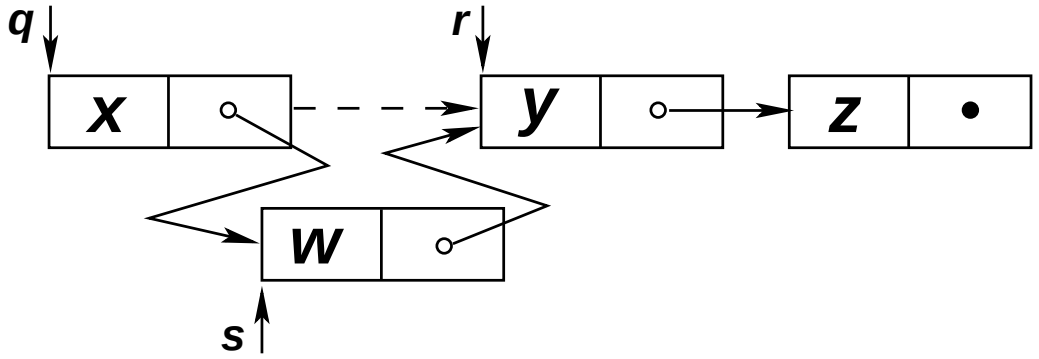


El tipo posición (cont.)

- Consideremos el proceso de insertar un elemento `w` en la posición en la que estaba `y`. Por lo tanto contamos con `r`
- Necesitamos hacer los siguientes enlaces

```
1  s = new cell;
2  s->elem = w;
3  s->next = r;
4  q->next = s;
```

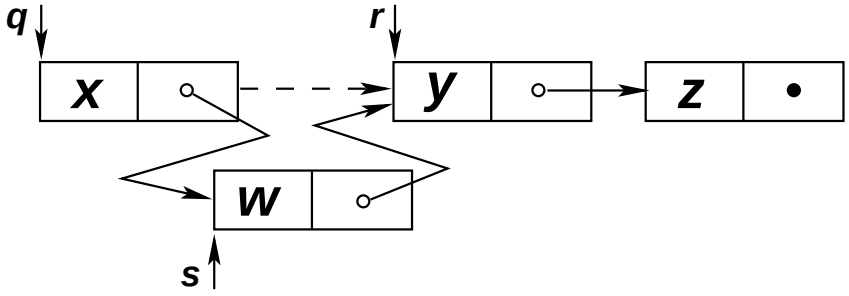
- Por lo que necesitamos `q`!!



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El tipo posición (cont.)

- **Definir al tipo posición como el puntero a la celda *anterior a la que contiene el dato*. Entonces tenemos, *antes* de insertar**
  - ▷ **`z` está en la posición `r`**
  - ▷ **`y` está en la posición `q`**
- **Después de insertar**
  - ▷ **`z` está en la posición `r`**
  - ▷ **`y` está en la posición `s`**
  - ▷ **`w` está en la posición `q`**



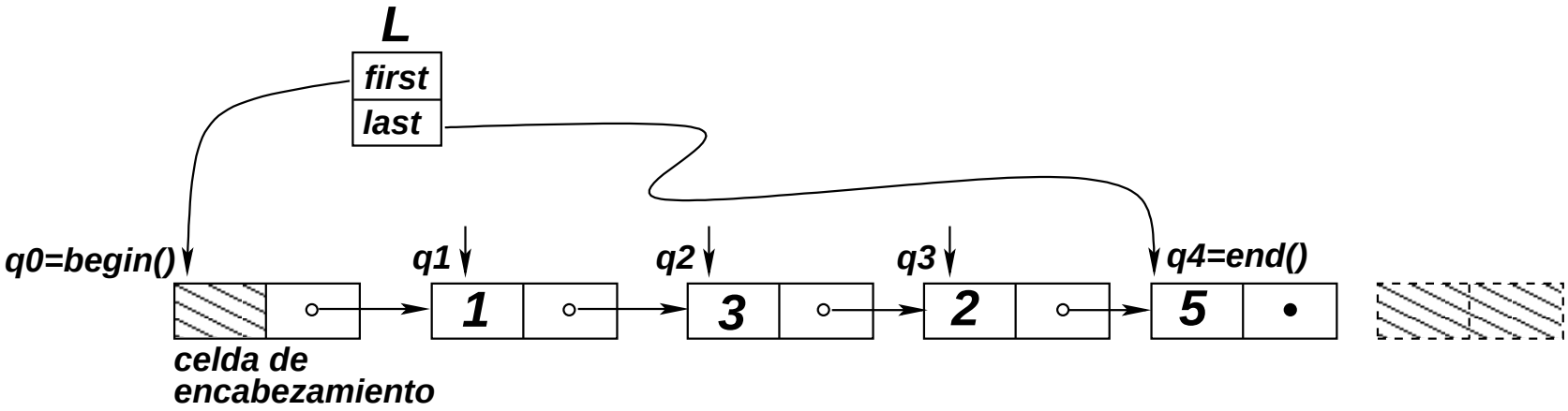
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El tipo posición (cont.)

- **Decimos que las posiciones están *"adelantadas"* con respecto al dato.**

Celda de encabezamiento

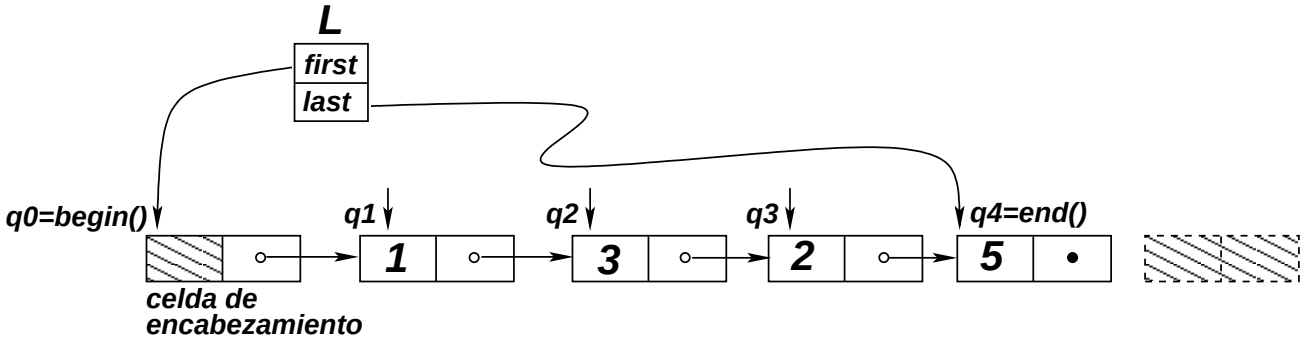
- ¿Cual es la posición del primer elemento? Introducimos una celda de *"encabezamiento"*.
- La lista en sí es un objeto que consiste en dos punteros uno a la primera celda (de encabezamiento) y otro a la última.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Celda de encabezamiento (cont.)

- `L.retrieve(q0)` retornará 1,
- `L.retrieve(q1)` retornará 3...
- `L.retrieve(q4)` dará error ya que corresponde a la posición no-dereferenciable `L.end()`



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

```
1  list::list() : first(new cell), last(first) {
2    first->next = NULL;
3  }
4
5  list::~list() { clear(); delete first; }
6
7  elem_t &list::retrieve(iterator_t p) {
8    return p->next->elem;
9  }
10
11 iterator_t list::next(iterator_t p) {
12   return p->next;
13 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

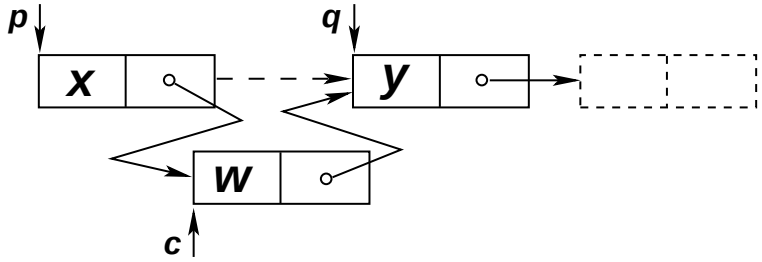
Implementación de listas con celdas enlazadas

```
1 iterator_t list::prev(iterator_t p) {
2   iterator_t q = first;
3   while (q->next != p) q = q->next;
4   return q;
5 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

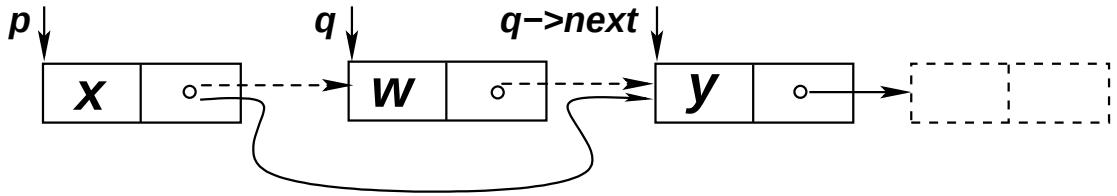
```
1  iterator_t
2  list::insert(iterator_t p,elem_t k) {
3    iterator_t q = p->next;
4    iterator_t c = new cell;
5    p->next = c;
6    c->next = q;
7    c->elem = k;
8    if (q==NULL) last = c;
9    return p;
10 }
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

```
1  iterator_t list::begin() { return first; }
2
3  iterator_t list::end() { return last; }
4
5  iterator_t list::erase(iterator_t p) {
6    if (p->next==last) last = p;
7    iterator_t q = p->next;
8    p->next = q->next;
9    delete q;
10   return p;
11 }
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

```
1  iterator_t list::erase(iterator_t p,iterator_t q) {
2    if (p==q) return p;
3    iterator_t s, r = p->next;
4    p->next = q->next;
5    if (!p->next) last = p;
6    while (r!=q->next) {
7      s = r->next;
8      delete r;
9      r = s;
10   }
11   return p;
12 }
```

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

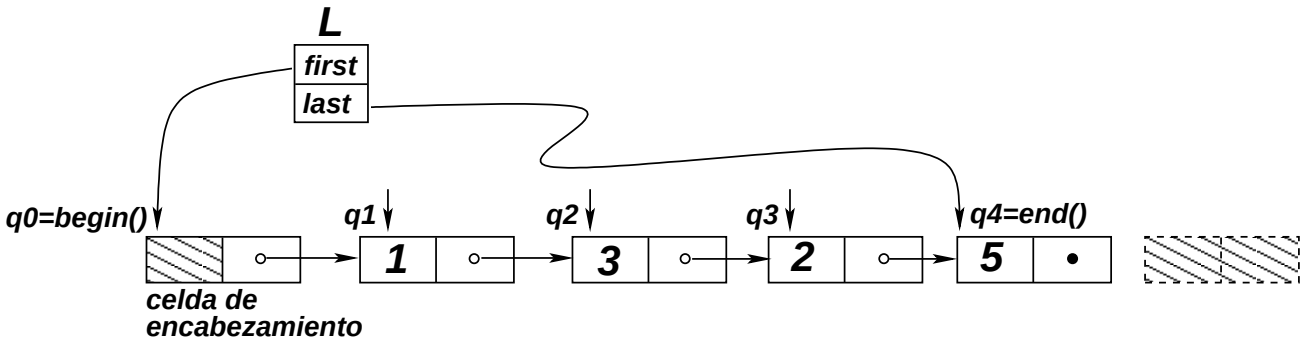
```
1  void list::clear() { erase(begin(),end()); }
2
3  void list::print() {
4    iterator_t p = begin();
5    while (p!=end()) {
6      cout << retrieve(p) << " ";
7      p = next(p);
8    }
9    cout << endl;
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

- `begin()` es un puntero a la celda de encabezamiento. No cambia nunca durante la vida de la lista.
- `end()` es un puntero a la última celda. En realidad no hace falta guardarlo en la lista, pero de no hacerlo la función `end()` debería implementarse, pero esto es $O(n)$. La solución es incluir un puntero a la última celda en el objeto lista.

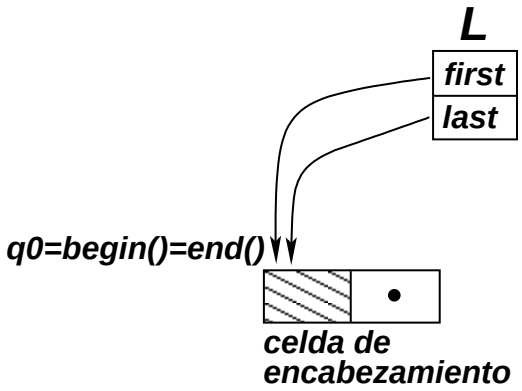
```
1  iterator_t list::end() {
2    cell *q = first;
3    while (q->next) q = q->next;
4    return q;
5  }
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de listas con celdas enlazadas

Lista vacía



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas por cursores

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas con celdas enlazadas por cursores

- **En la implementación previa las celdas son alocadas en el área de almacenamiento dinámico del programa. Una variante de esto es usar celdas indexadas por enteros dentro de un gran arreglo de celdas.**
- **Es en muchos aspectos similar a la implementación por punteros. Las funciones pueden *"traducirse"* de punteros a cursores y viceversa.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interés de la implementación por cursores

- **La gestión de celdas puede llegar a ser más eficiente.**
- **Alocación de muchos pequeños objetos con `new` y `delete` puede fragmentar la memoria, lo cual puede impedir la alocación de objetos grandes. Usando cursores, las celdas no se mezclan con el resto de los objetos del programa.**
- **Reemplaza a los punteros en lenguajes donde no existen éstos.**
- **El estudio de los cursores ayuda a entender cómo funciona el área de alocamiento dinámica (*"heap"* o *"free store"*).**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Desventajas de los cursores

- **Hay que reservar un gran arreglo de celdas. Si reservamos pocas, después se pueden acabar. Si reservamos muchas, estamos alocando memoria que en la práctica no será usada.**
- **Listas de elementos del mismo tipo comparten el mismo espacio pero cada tipo requiere su propio espacio, agravando el problema anterior.**

Declaraciones de listas por cursores

```
1   class list;
2   typedef int iterator_t;
3
4   class cell {
5     friend class list;
6     elem_t elem;
7     iterator_t next;
8     cell();
9   };
10
11  class list {
12  private:
13    friend class cell;
14    static iterator_t NULL_CELL;
15    static int CELL_SPACE_SIZE;
16    static cell *cell_space;
17    static iterator_t top_free_cell;
18    iterator_t new_cell();
19    void delete_cell(iterator_t c);
20    iterator_t first, last;
21    void cell_space_init();
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas por cursores

- Las celdas son como en el caso de los punteros, pero ahora el campo `next` es de tipo entero, así como las posiciones (`iterator_t`).
- Las celdas viven en el arreglo `cell_space`, que es un arreglo estándar de elementos de tipo `cell`.

```
1   class cell {
2     friend class list;
3     elem_t elem;
4     iterator_t next;
5     cell();
6   };
7
8   class list {
9   private:
10    static int CELL_SPACE_SIZE;
11    static cell *cell_space;
12    ....
```

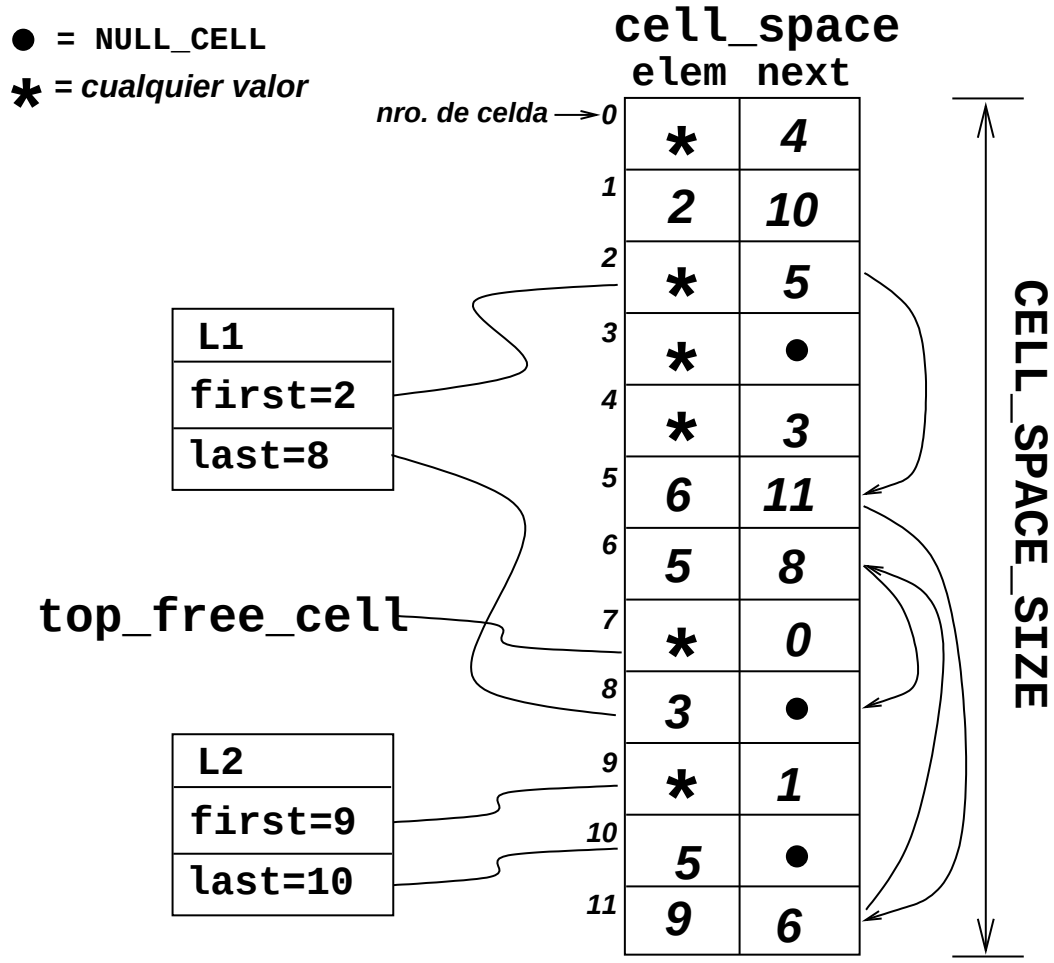

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas por cursores

- `cell_space` es declarado `static`, de esta forma actúa como si fuera global, pero dentro de la clase.
- También declaramos `static` el tamaño del arreglo `CELL_SPACE_SIZE`.
- Así como con punteros existe el puntero inválido `NULL`, declaramos un cursor inválido `NULL_CELL=-1`.

```
1
2  class list {
3  private:
4    ...
5    static int CELL_SPACE_SIZE;
6    static iterator_t NULL_CELL;
7    static iterator_t top_free_cell;
8    ...
```

Listas por cursores



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Gestión de celdas

- **Debemos escribir funciones que emulen**
  - ▷ `new` ⟶ `new_cell()`
  - ▷ `delete` ⟶ `delete_cell()`
- **Mantener una lista de celdas enlazadas con las celdas libres**
- `top_free_cell` **apunta a la primera celda libre**
- **El campo** `next` **de la i-ésima celda libre apunta a la (i+1)-ésima celda libre. La última celda libre tiene un campo** `next` **inválido (**`NULL_CELL`**) .**
- **Es como una lista normal enlazada pero sin posiciones. (En realidad es una pila).**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Gestión de celdas (cont.)

- Las rutinas `c=new_cell()` y `delete_cell(c)` definen una interfase abstracta (dentro de la clase `list`) para la gestión de celdas.

```
1  iterator_t list::new_cell() {
2    iterator_t top = top_free_cell;
3    if (top==NULL_CELL) {
4      cout << "No hay mas celdas \n";
5      abort();
6    }
7    top_free_cell = cell_space[top_free_cell].next;
8    return top;
9  }
10
11 void list::delete_cell(iterator_t c) {
12   cell_space[c].next = top_free_cell;
13   top_free_cell = c;
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Gestión de celdas

- Antes de hacer cualquier operación sobre una lista hay que asegurarse que el espacio de celdas este inicializado.
- Esto se hace en `cell_space_init()` que aloca `cell_space` y crea la lista de celdas libres.
- `cell_space_init()` lo incluimos en el constructor de la clase lista.

```
1  list::list() {
2    if (!cell_space) cell_space_init();
3    first = last = new_cell();
4    cell_space[first].next = NULL_CELL;
5  }
6
7  void list::cell_space_init() {
8    cell_space = new cell[CELL_SPACE_SIZE];
9    for (int j=0; j<CELL_SPACE_SIZE-1; j++)
10     cell_space[j].next = j+1;
11   cell_space[CELL_SPACE_SIZE-1].next = NULL_CELL;
12   top_free_cell = 0;
13 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

	Punteros	Cursores
Area de almacenamiento	heap	`cell_space`
Tipo para direcciones de las celdas	`cell* c`	`int c`
Dereferenciación de direcciones (dirección $\longrightarrow$ celda)	`*c`	`cell_space[c]`
Dato de una celda dada su dirección `c`	`c->elem`	`cell_space[c].elem`
Enlace de una celda (campo `next`) dada su dirección `c`	`c->next`	`cell_space[c].next`
Alocar una celda	`c = new cell;`	`c = new_cell();`
Liberar una celda	`delete c;`	`delete_cell(c);`
Dirección inválida	`NULL`	`NULL_CELL`

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

insert() por punteros y cursores

```
1  iterator_t
2  list::insert(iterator_t p,elem_t k) {
3    iterator_t q = p->next;
4    iterator_t c = new cell;
5    p->next = c;
6    c->next = q;
7    c->elem = k;
8    if (q==NULL) last = c;
9    return p;
10 }
```

```
1  iterator_t list::insert(iterator_t p,elem_t k) {
2    iterator_t q = cell_space[p].next;
3    iterator_t c = new_cell();
4    cell_space[p].next = c;
5    cell_space[c].next = q;
6    cell_space[c].elem = k;
7    if (q==NULL_CELL) last = c;
8    return p;
9  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución

Método	Arreglos	Punteros / cursores
`insert(p,x)`, `erase(p)`	$O(n)$ $[T = c(n-j)]$	$O(1)$
`erase(p,q)`	$O(n)$ $[T = c(n-k)]$	$O(n)$ $[T = c(k-j)]$
`clear()`	$O(1)$	$O(n)$
`begin()`, `end()`, `next()`, `retrieve()`	$O(1)$	$O(1)$
`prev(p)`	$O(1)$	$O(n)$ $[T = cj]$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase STL para lista

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase STL

- En la forma en que hemos visto, la lista sólo puede tener un tipo de elemento dado, listas de enteros o de dobles, pero no de las dos a un mismo tiempo. El tipo se puede definir insertando una línea `typedef elem_t int` al principio.
- Si queremos manipular listas de diferente tipo, entonces se debe duplicar el código y definir tipos `list_int` y `list_double`, lo mismo con los iterators `list_int_iterator` y `list_double_iterator`.
- C++ ofrece la posibilidad de los templates

```
1 list<int> lista_1;
2 list<double> lista_2;
```

Sobrecarga de operadores

```
p = next(p);   ⟶  p++
p = prev(p);   ⟶  p--
x = L.retrieve(p);   ⟶  x = *p
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Clases anidadas para los iteradores

Podemos también definir a los iterators como templates `iterator<int> p,q`. Pero ¿Que pasará cuando tengamos otros contenedores, como `vector<>`, `set<>`? Necesitaremos un tipo `list_iterator<int>`, `set_iterator<int>`... Para evitar esto definimos a la clase anidada, es decir dentro del contenedor

```
1 template<class T>
2 class list {
3   friend class iterator;
4   ...
5   class iterator {
6     friend class list;
7     ...
8   }
9 }
```

de esta forma la clase del contenedor actúa como un *"namespace"* para la clase anidada del iterator y (fuera de la clase `list<>`) habrá que llamarla como `list<int>::iterator`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Clases anidadas para los iteradores

El anidamiento (*"nesting"*) de clases no tiene nada que ver con la amistad (*"friendship"*) entre ellas. De manera que debemos declarar en cada una de ellas como `friend` a la otra.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operador ++ prefijo y postfijo

Recordemos que los operadores de incremento *"prefijo"* (`++p`) y *"postfijo"* (`p++`) tienen el mismo *"efecto colateral"* (*"side effect"*) que es incrementar la variable `p` pero tienen diferente *"valor de retorno"* a saber el valor incrementado en el caso del postfijo y el valor no incrementado para el prefijo. Es decir

- `q = p++;` **es equivalente a** `q = p; p = p.next();`**, mientras que**
- `q = ++p;` **es equivalente a** `p = p.next(); q = p;` **.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template list<class T>

```
1  namespace aed {
2
3    template<class T>
4    class list {
5    public:
6      class iterator;
7    private:
8      class cell {
9        friend class list;
10       friend class iterator;
11       T t;
12       cell *next;
13       cell() : next(NULL) {}
14     };
15     cell *first, *last;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
1   public:
2     class iterator {
3     private:
4       friend class list;
5       cell* ptr;
6     public:
7       T & operator*() { return ptr->next->t; }
8       T *operator->() { return &ptr->next->t; }
9       bool operator!=(iterator q) { return ptr!=q.ptr; }
10      bool operator==(iterator q) { return ptr==q.ptr; }
11      iterator(cell *p=NULL) : ptr(p) {}
12      // Prefix:
13      iterator operator++() {
14        ptr = ptr->next;
15        return *this;
16      }
17      // Postfix:
18      iterator operator++(int) {
19        iterator q = *this;
20        ptr = ptr->next;
21        return q;
22      }
23    };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template `list<class T>`

```
1     list() {
2       first = new cell;
3       last = first;
4     }
5     iterator insert(iterator p,T t) {
6       cell *q = p.ptr->next;
7       cell *c = new cell;
8       p.ptr->next = c;
9       c->next = q;
10      c->t = t;
11      if (q==NULL) last = c;
12      return p;
13    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template list<class T>

```
1   iterator erase(iterator p) {
2     cell *q = p.ptr->next;
3     if (q==last) last = p.ptr;
4     p.ptr->next = q->next;
5     delete q;
6     return p;
7   }
8   iterator erase(iterator p,iterator q) {
9     cell *s, *r = p.ptr->next;
10    p.ptr->next = q.ptr->next;
11    if (!p.ptr->next) last = p.ptr;
12    while (r!=q.ptr->next) {
13      s = r->next;
14      delete r;
15      r = s;
16    }
17    return p;
18  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template `list<class T>`

```
1    void clear() { erase(begin(),end()); }
2    iterator begin() { return iterator(first); }
3    iterator end() { return iterator(last); }
4    void print() {
5      iterator p = begin();
6      while (p!=end()) std::cout << *p++ << " ";
7      std::cout << std::endl;
8    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template `list<class T>`

- **Hemos incluido un `namespace aed`. Por lo tanto las clases deben ser referenciadas como `aed::list<int>` y `aed::list<int>::iterator`. O usar: `using namespace aed;`.**
- **`template<class T>` indica que la clase `T` es un tipo genérico. Al hacer `list<int>` `T` pasa a tomar el valor concreto `int`.**
- **`iterator<T>` y `cell<T>` son clases *anidadas* dentro de la clase `list<T>`. Evita tener que agregarle un prefijo o sufijo (como en `list_cell`, `stack_cell`, etc...).**
- **`cell<T>` declara `friend` a `list<T>` e `iterator<T>`. (Recordar que anidamiento no implica amistad). Si queremos que `list<T>` acceda a los miembros *privados* de `cell<T>` e `iterator<T>` entonces debemos declarar a las `list` como `friend.` en `cell` e `iterator`.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El template `list<class T>`

- Las clases `list<T>`, `cell<T>` e `iterator<T>` son un ejemplo de *"clases fuertemente ligadas"* (*"tightly coupled classes"*).
- La clase `cell<T>` es declarada privada dentro de `list<T>` ya que normalmente *no debe ser accedida por el usuario de la clase*.
- Para poder sobrecargar los operadores de incremento y dereferenciación (`p++`, `++p` y `*p`) debemos declarar a `iterator<T>` como una clase y no como un `typedef` como en la interfase básica. `iterator` contiene un `cell *ptr` de manera que

```
1 iterator_t q = p->next;
```

se convierte en

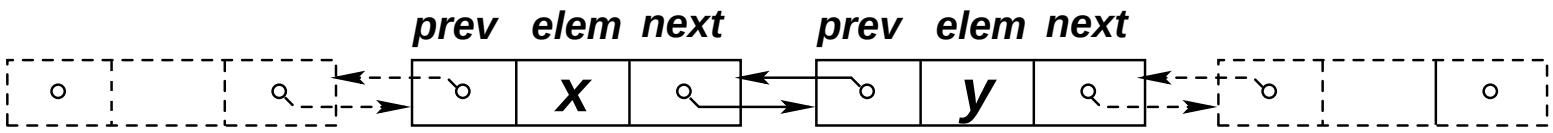
```
1 cell *q = p.ptr->next;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas doblemente enlazadas

Si es necesario utilizar repetidamente `p--` (`p=p.prev()`) entonces conviene usar una lista doblemente enlazada.

```
1 class cell {
2   elem_t elem;
3   cell *next, *prev;
4   cell() : next(NULL), prev(NULL) {}
5 };
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listas doblemente enlazadas (cont.)

- **No es necesario adelantar las posiciones. (La posición es el puntero a *la celda que contiene al elemento*).**
- **La clase `list<>` de STL es en realidad doblemente enlazada. La clase `slist<>` es simplemente enlazada.**

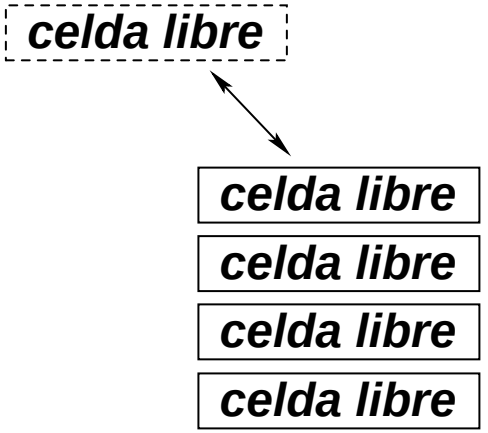
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD pila

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD pila

- **Es en realidad un subtipo de la lista, donde todas las operaciones se realizan en un extremo de la lista: el *tope*.**
- **Se le llama estructura *"LIFO"* (*"Last In First Out"*), es decir *"el último en entrar es el primero en salir"*).**
  - **Un ejemplo de la pila es la lista enlazada de celdas libres que usamos en la implementación de listas por cursores.**



- **En el apunte hay un ejemplo más complejo de como escribir una calculadora basada en una pila.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones abstractas sobre pilas

El ejemplo de las celdas libres sugiere las siguientes operaciones abstractas

- **Insertar un elemento en el tope de la pila.**
- **Obtener el valor del elemento en el tope de la pila.**
- **Eliminar el elemento del tope.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase para pila

```
1 elem_t top();
2 void pop();
3 void push(elem_t x);
4 void clear();
5 int size();
6 bool empty();
```

- `x = top()` **devuelve el elemento en el tope de la pila (sin modificarla).**
- `pop()` **remueve el elemento del tope (sin retornar su valor!).**
- `push(x)` **inserta el elemento** `x` **en el tope de la pila.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase para pila

- **Esta interfase es directamente compatible con STL.**
- **Casi todas las librerías que implementan pilas usan una interfase similar a esta.**
- **Hemos agregado**
  - ▷ `void clear()` **remueve todos los elementos de la pila.**
  - ▷ `int size()` **devuelve el número de elementos en la pila.**
  - ▷ `bool empty()` **retorna verdadero si la pila esta vacía, verdadero en caso contrario. (Notar que** `empty()` ***no modifica*** **la pila, mucha gente tiende a confundirla con** `clear()`**.)**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

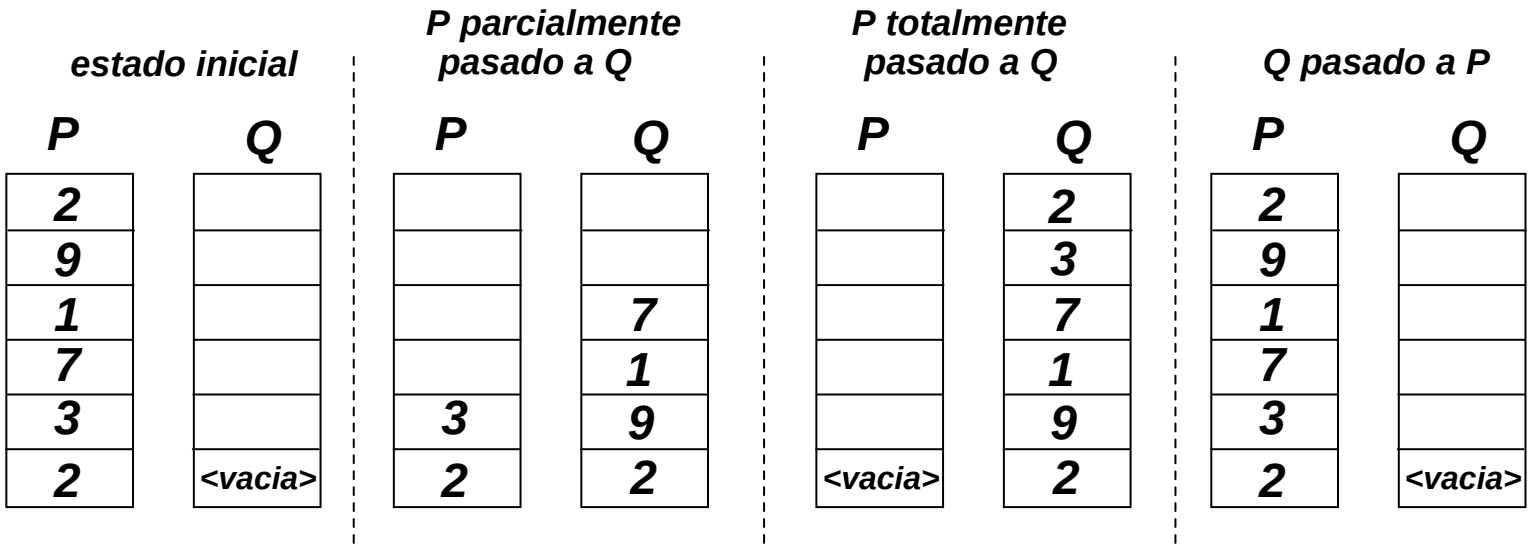
Ejemplo. Sumar los elementos de una pila

```
1 int sumstack(stack<int> &P) {
2   int sum=0;
3   while (!P.empty()) {
4     sum += P.top();
5     P.pop();
6   }
7   return sum;
8 }
```

Esta suma es *destructiva* ya que la pila queda vacía, después de haberla sumado.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo. Sumar los elementos de una pila



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo. Sumar los elementos de una pila

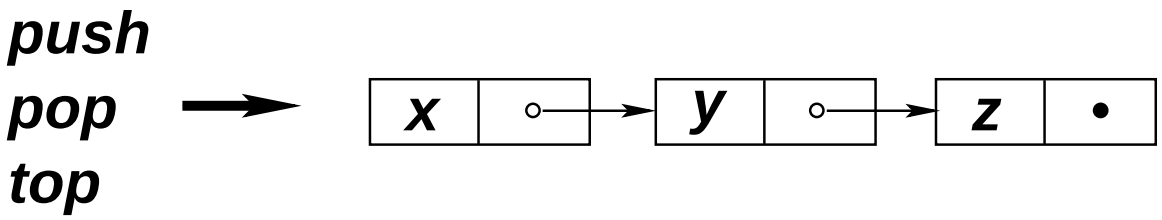
Versión no *destructiva*:

```
1  int sumstack2(stack<int> &P) {
2    stack<int> Q;
3    int sum=0;
4    while (!P.empty()) {
5      int w = P.top();
6      P.pop();
7      sum += w;
8      Q.push(w);
9    }
10
11   while (!Q.empty()) {
12     P.push(Q.top());
13     Q.pop();
14   }
15   return sum;
16 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de pilas mediante listas

- **La pila se implementa a partir de una lista, *asumiendo que el tope de la pila está en el comienzo de la lista*.**
- **`push(x)` y `pop()` se pueden implementar a partir de `insert` y `erase` en el comienzo de la lista.**
- **`top()` se puede implementar en base a `retrieve`.**
- **La implementación por punteros y cursores es apropiada, ya que todas estas operaciones son $O(1)$. Notar que si se usa el último elemento de la lista como tope, entonces las operaciones de `pop()` pasan a ser $O(n)$.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Pilas mediante listas. Declaraciones

```
1   class stack : private list {
2   private:
3     int size_m;
4   public:
5     stack();
6     void clear();
7     elem_t& top();
8     void pop();
9     void push(elem_t x);
10    int size();
11    bool empty();
12  };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Pilas mediante listas. Implementación

```
1  stack::stack() : size_m(0) { }
2
3  elem_t& stack::top() {
4    return retrieve(begin());
5  }
6
7  void stack::pop() {
8    erase(begin()); size_m--;
9  }
10
11 void stack::push(elem_t x) {
12   insert(begin(),x); size_m++;
13 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

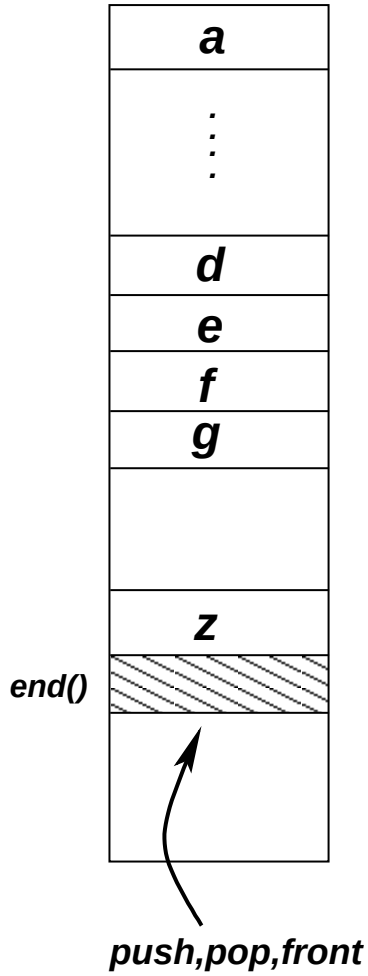
Pilas mediante listas. Implementación

```
1  void stack::clear() {
2    erase(begin(),end()); size_m = 0;
3  }
4
5  bool stack::empty() {
6    return begin()==end();
7  }
8
9  int stack::size() {
10   return size_m;
11 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de pilas por arreglos.

- **Tanto la inserción como la supresión en el primer elemento son $O(n)$.**
- **Tanto la inserción como la supresión en el último elemento son $O(1)$.**
- **De manera que es posible implementar una clase de pila basada en arreglos.**



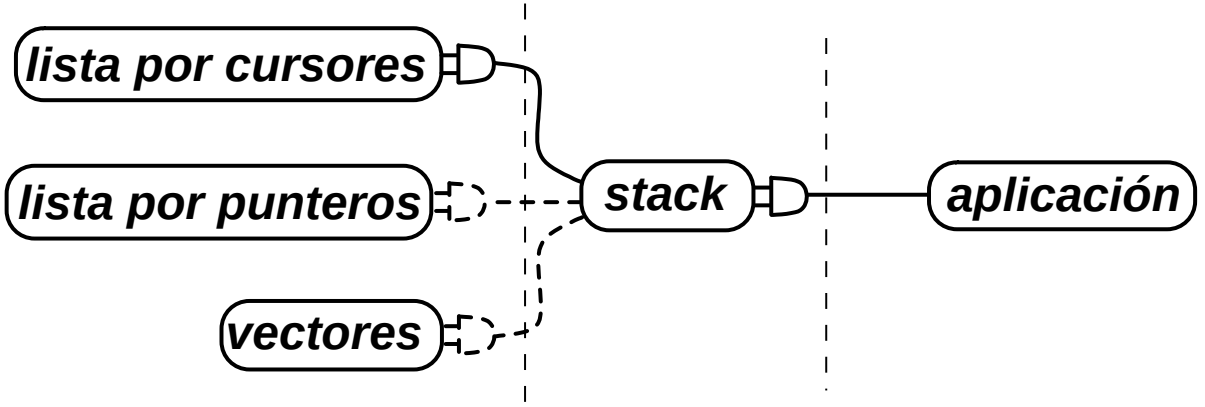
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tamaño de la pila

El tamaño de la pila es guardado en un miembro `int size_m`. Este contador es inicializado a cero en el constructor y después es actualizado durante las operaciones que modifican la longitud de la lista como `push()`, `pop()` y `clear()`.

La pila como adaptador

- **Notar que la pila deriva directamente de la lista, pero con una declaración `private`. De esta forma el usuario de la clase `stack` no puede usar métodos de la clase lista.**
- **El hecho de que la pila sea tan simple permite que pueda ser implementada en términos de otros contenedores también, como por ejemplo el contenedor `vector` de STL.**
- **De esta forma, podemos pensar a la pila como un *"adaptador"* (*"container adaptor"*).**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase STL

La única diferencia con la interfase STL es el uso de templates.

```
1  template<class T>
2  class stack : private list<T> {
3  private:
4    int size_m;
5  public:
6    stack() : size_m(0) { }
7    void clear() { erase(begin(),end()); size_m = 0; }
8    T &top() { return *begin(); }
9    void pop() { erase(begin()); size_m--; }
10   void push(T x) { insert(begin(),x); size_m++; }
11   int size() { return size_m; }
12   bool empty() { return size_m==0; }
13 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD cola

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD cola

- **Por contraposición con la pila, la cola es un contenedor de tipo *"FIFO"* (por *"First In First Out"*, el primero en entrar es el primero en salir).**
- **La cola es un objeto muchas veces usado como buffer o pulmón, es decir un contenedor donde almacenar una serie de objetos que deben ser procesados, manteniendo el orden en el que ingresaron.**
- **La cola es, como la pila, un subtipo de la lista y llama también a ser implementada como un *adaptador*.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones abstractas sobre cola

- **Obtener el elemento en el frente de la cola**
- **Eliminar el elemento en el frente de la cola**
- **Agregar un elemento a la cola**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase para cola

```
1  template<class T>
2  class queue : private list<T> {
3  private:
4    int size_m;
5  public:
6    queue() : size_m(0) { }
7    void clear() { erase(begin(),end()); size_m = 0; }
8    T front() { return *begin(); }
9    void pop() { erase(begin()); size_m--; }
10   void push(T x) { insert(end(),x); size_m++; }
11   int size() { return size_m; }
12   bool empty() { return size_m==0; }
13 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Colas. Ejemplo de programación

Consigna: **Escribir un programa que calcula la suma de los elementos de una cola.**

Solución: **Igual que con la pila hay una solución destructiva y otra no.**

- **Inicialmente:**
  $Q = (2, 3, 5, 4, 3, 1)$, $Q2 = ()$.
- **Después de haber pasado algunos elementos de Q a $Q2$:**
  $Q = (4, 3, 1)$, $Q2 = (2, 3, 5)$.
- **Después de haber pasado todos los elementos de Q a $Q2$:**
  $Q = ()$, $Q2 = (2, 3, 5, 4, 3, 1)$.
- **Después de volver todos los elementos de $Q2$ a Q:**
  $Q = (2, 3, 5, 4, 3, 1)$, $Q2 = ()$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

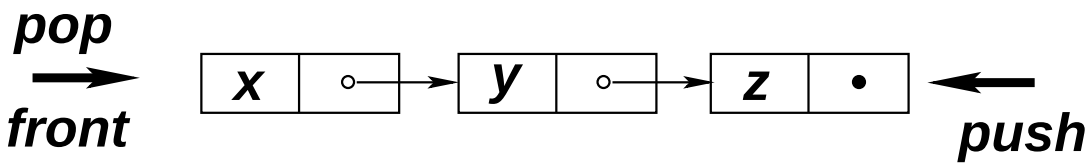
Colas. Ejemplo de programación

```
1  int sumqueue(queue<int> &Q) {
2    int sum=0;
3    queue<int> Q2;
4    while (!Q.empty()) {
5      int w = Q.front();
6      Q.pop();
7      sum += w;
8      Q2.push(w);
9    }
10
11   while (!Q2.empty()) {
12     Q.push(Q2.front());
13     Q2.pop();
14   }
15   return sum;
16 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de colas basada en listas

- Igual que para la pila, la cola no tiene posiciones, de manera que no necesita sobrecarga de operadores ni clases anidadas.
- `pop()` y `front()` actúan sobre el frente de la cola (el principio de la lista).
- `push()` actúa sobre el fondo de la cola (el final de la lista).



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución

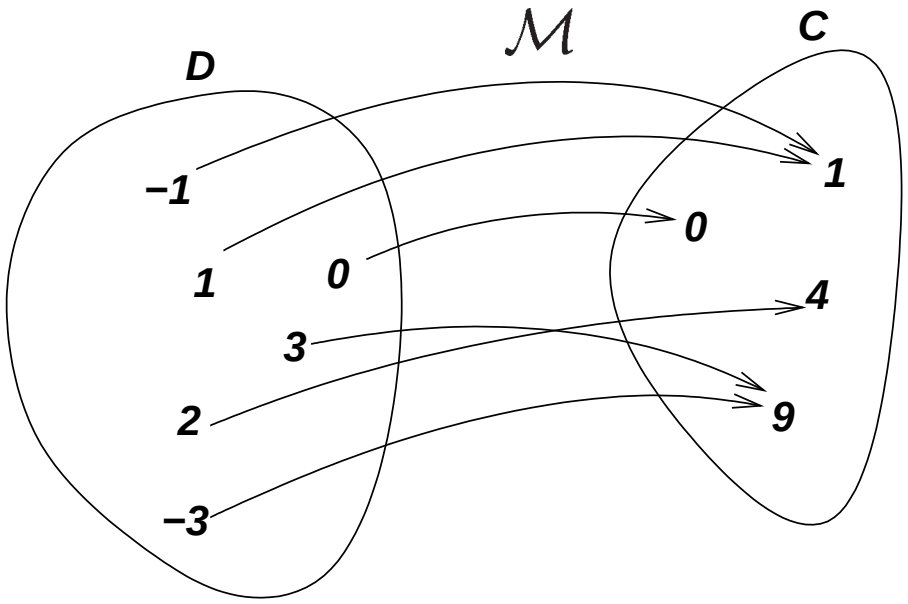
- **`pop()`, `front()`, y `push()` son $O(1)$. Notar que si hubiéramos invertido los extremos (frente de cola=fin de la lista y fondo de la cola = comienzo de la lista), entonces `pop()` sería $O(n)$ para listas simplemente enlazadas.**
- **También hemos agregado, como en el caso de la pila operaciones estándar `size()` y `empty()`, que también son $O(1)$ y `clear()` que es $O(n)$.**

Correspondencias

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD correspondencia

- **La *"correspondencia"* o *"memoria asociativa"* almacena la relación entre elementos de un cierto conjunto universal D llamado el *"dominio"* con elementos de otro conjunto universal llamado el *"contradominio"* o *"rango"*.**
  - **Por ejemplo $j \rightarrow j^2$ de enteros a enteros.**
  - **La correspondencia debe ser *"unívoca"*.**
  - **Al elemento del dominio se le llama a veces *"clave"* y al del contradominio *"valor"***



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El TAD correspondencia (cont.)

- **En general se guardan internamente pares clave/valor y poseen algún algoritmo para asignar valores a claves, en forma análoga a como funcionan las bases de datos para luego recuperar rápidamente la asignación.**
- **Para $j \longrightarrow j^2$, es mucho más eficiente usar una función**
- **El uso de un contenedor tipo correspondencia es útil justamente cuando no es posible calcular el elemento del contradominio a partir del elemento del dominio. Por ejemplo, un tal caso es una correspondencia entre el número de documento de una persona y su nombre. Es imposible *"calcular"* el nombre a partir del número de documento.**

Ejemplo de uso del TAD correspondencia

Mantener una lista de sueldos. Consigna: **Escribir un programa que memoriza para cada documento de identidad el sueldo de un empleado. Se van ingresando números de documento. Si el documento ya tiene un sueldo asignado, se reporta, sino el usuario debe entrar un sueldo.**

```
1  [mstorti@spider aedsrc]$ payroll
2  Ingrese nro. documento > 14203323
3  Ingrese salario mensual: 2000
4  Ingrese nro. documento > 13324435
5  Ingrese salario mensual: 3000
6  Ingrese nro. documento > 13323421
7  Ingrese salario mensual: 2500
8  Ingrese nro. documento > 14203323
9  Doc: 14203323, salario: 2000
10 Ingrese nro. documento > 13323421
11 Doc: 13323421, salario: 2500
12 Ingrese nro. documento > 13242323
13 Ingrese salario mensual: 5000
14 Ingrese nro. documento > 0
15 No se ingresan mas sueldos...
16 [mstorti@spider aedsrc]$
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso del TAD correspondencia (cont.)

```
1   // declarar 'tabla_sueldos' como 'map' . . .
2   while(1) {
3     cout << "Ingrese nro. documento > ";
4     int doc;
5     double sueldo;
6     cin >> doc;
7     if (!doc) break;
8     if (/* No tiene 'doc' sueldo asignado?. . . */) {
9       cout << "Ingrese sueldo mensual: ";
10      cin >> sueldo;
11      // Asignar 'doc -> sueldo'
12      // . . .
13    } else {
14      // Reportar el valor almacenado
15      // en 'tabla_sueldos'
16      // . . .
17      cout << "Doc: " << doc << ", sueldo: "
18           << sueldo << endl;
19    }
20  }
21  cout << "No se ingresan mas sueldos. . ." << endl;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones abs. sobre el TAD correspondencia

- **Consultar la correspondencia para saber si una dada clave tiene un valor asignado.**
- **Asignar un valor a una clave.**
- **Recuperar el valor asignado a una clave.**

Interfase simple para correspondencias

```
1  class iterator_t {/* . . . */};
2
3  class map {
4   private:
5    // . . .
6   public:
7    iterator_t find(domain_t key);
8    iterator_t insert(domain_t key,range_t val);
9    range_t& retrieve(domain_t key);
10   void erase(iterator_t p);
11   int erase(domain_t key);
12   domain_t key(iterator_t p);
13   range_t& value(iterator_t p);
14   iterator_t begin();
15   iterator_t next(iterator_t p);
16   iterator_t end();
17   void clear();
18   void print();
19 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase simple para correspondencias (cont.)

- **Se deben definir (probablemente via `typedef`'s) los tipos que corresponden al domininio (`domain_t`) y al contradominio (`range_t`).**
- **Una clase `iterator` representa las posiciones en la correspondencia. En la correspondencia el iterator *itera sobre los pares de valores* que representan la correspondencia.**
- **Notar que, en contraposición con las listas y a semejanza de los conjuntos, no hay un orden definido entre los pares de la correspondencia.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase simple para correspondencias (cont.)

`M` es una correspondencia, `p` es un iterator, `k` una clave, `val` un elemento del contradominio (tipo `range_t`)

- **`p = M.find(k)`: Dada una clave `k` devuelve un iterator *al par correspondiente* (si existe debe ser único). Si `k` no tiene asignado ningún valor, entonces devuelve `end()`.**
- **`p = M.insert(k,val)`: asigna a `k` el valor `val`. Si `k` ya tenía asignado un valor, entonces este nuevo valor reemplaza a aquel en la asignación. Si `k` no tenía asignado ningún valor entonces la nueva asignación es definida. Retorna un iterator al par.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase simple para correspondencias (cont.)

- `val = M.retrieve(k)`: Recupera el valor asignado a `k`. Si `k` no tiene ningún valor asignado, entonces *inserta una asignación de `k` al valor creado por defecto* para el tipo `range_t` (es decir el que retorna el constructor `range_t()`). *Esto es muy importante y muchas veces es fuente de error.* (Muchos esperan que `retrieve()` de un error en ese caso.) Si queremos recuperar en `val` el valor asignado a `k` *sin insertar accidentalmente una asignación* en el caso que `k` no tenga asignado ningún valor entonces debemos hacer

```
1  if (M.find(k)!=M.end()) val = M.retrieve(k);
```

  `M.retrieve(k)` retorna una *referencia* al valor asignado a `k`, de manera que también puede ser usado como miembro izquierdo, es decir, es válido hacer

```
1  M.retrieve(k) = val;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase simple para correspondencias (cont.)

- `k = M.key(p)` retorna el valor correspondiente a la asignación *apuntada* por `p`.
- `val = M.value(p)` retorna el valor correspondiente a la asignación *apuntada* por `p`. El valor retornado es una referencia, de manera que también podemos usar `value(p)` como miembro izquierdo (es decir, asignarle un valor como en `M.value(p)=val`). Notar que, por el contrario, `key(p)` no retorna una referencia.
- `erase(p)`: Elimina la asignación apuntada por `p`. Si queremos eliminar una eventual asignación a la clave `k` entonces debemos hacer

```
1  p = M.find(k);
2  if (p!=M.end()) M.erase(p);
```

- `n = erase(k)`: Elimina la asignación correspondiente a `k` (si la hay). Retorna el número de asignaciones efectivamente eliminadas (0 o 1).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase simple para correspondencias (cont.)

- `p = M.begin()`: Retorna un iterator a la primera asignación (en un orden no especificado).
- `p = M.end()`: Retorna un iterator a una asignación ficticia después de la última (en un orden no especificado).
- `clear()`: Elimina todas las asignaciones.
- `print()`: Imprime toda la tabla de asignaciones. (No existe en las STL!!)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del ejemplo

```
1   map sueldo;
2   while(1) {
3     cout << "Ingrese nro. documento > ";
4     int doc;
5     double salario;
6     cin >> doc;
7     if(!doc) break;
8     iterator_t q = sueldo.find(doc);
9     if (q==sueldo.end()) {
10      cout << "Ingrese salario mensual: ";
11      cin >> salario;
12      sueldo.insert(doc,salario);
13      cout << sueldo.size() << " salarios cargados" << endl;
14    } else {
15      cout << "Doc: " << doc << ", salario: "
16           << sueldo.retrieve(doc) << endl;
17    }
18  }
19  cout << "No se ingresan mas sueldos..." << endl;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación del ejemplo (cont.)

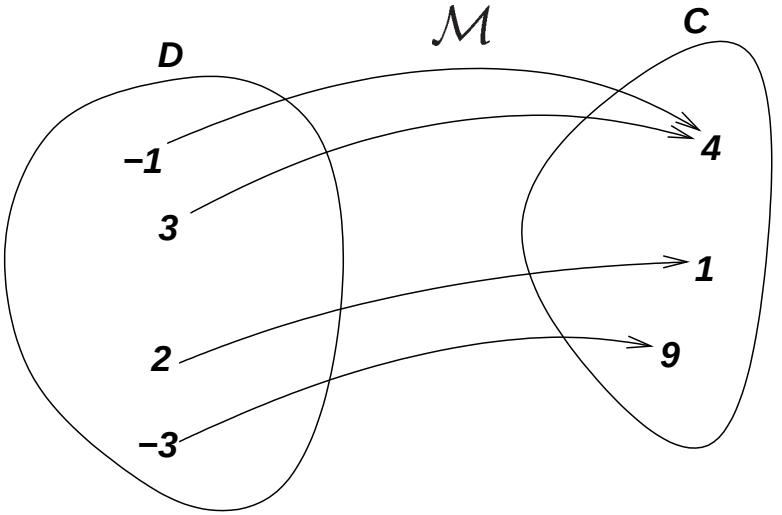
- **Notar que el programa es cuidadoso en cuanto a no crear nuevas asignaciones.**
- **El `retrieve` está garantizado que no generará ninguna asignación involuntaria ya que el test del `if` garantiza que `doc` ya tiene asignado un valor.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Correspondencias con contenedores lineales

- **Guardar en un contenedor todas las asignaciones.**

- **Definimos una clase `elem_t` que simplemente contiene dos campos `first` y `second` con la clave y el valor de la asignación (los nombres de los campos vienen de la clase `pair` de las STL).**



- **Estos pares de elementos (asignaciones) se podrían guardar tanto en un `vector<elem_t>` como en una lista (`list<elem_t>`). Por ejemplo, la correspondencia de enteros a enteros que se muestra en la figura se puede representar almacenando los siguientes pares en un contenedor: (-1,4),(3,4),(2,1),(-3,9)**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Correspondencias con contenedores lineales (cont.)

- **Discutiremos el TAD correspondencia basado en contenedores lineales (listas y vectores). Más adelante, veremos otras implementaciones más eficientes.**
- **Cuando hablemos de la listas asumiremos listas por punteros o cursores**
- **Para vectores asumiremos arreglos estándar de `C++` o `vector` de STL.**
- **`p=find(k)` debe recorrer todas las asignaciones y si encuentra una cuyo campo `first` coincida con `k` entonces debe devolver el iterator correspondiente. Si la clave no tiene ninguna asignación, entonces después de recorrer todas las asignaciones, debe devolver `end()`. El peor caso de `find()` es cuando la clave no está asignada, o la asignación está al final del contenedor, en cuyo caso es $O(n)$ ya que debe recorrer todo el contenedor (n es el número de asignaciones en la correspondencia).**
  Ej: M = ((-1,4),(3,4),(2,1),(-3,9))

Correspondencias con contenedores lineales (cont.)

- **Si la clave tiene una asignación, entonces el costo es proporcional a la distancia desde la asignación hasta el origen.**
- **En el peor caso esto es $O(n)$, como ya mencionamos, mientras que en el mejor caso, que es cuando la asignación está al comienzo del contenedor, es $O(1)$.**
- **La distancia media de la asignación es la mitad del número de asociaciones y por lo tanto en promedio el costo será $O(n/2)$.**
- **`insert()` debe llamar inicialmente a `find()`. Si la clave ya tiene un valor asignado, la inserción es $O(1)$ tanto para listas como vectores.**
- **En el caso de vectores, notar que esto se debe a que no es necesario insertar una nueva asignación. Si la clave no está asignada entonces el elemento se puede insertar al final para vectores, lo cual también es $O(1)$, y en cualquier lugar para listas.**

Correspondencias con contenedores lineales (cont.)

- **Para `erase(p)` sí hay diferencias, la implementación por listas es $O(1)$ mientras que la implementación por vectores es $O(n)$ ya que implica mover elementos.**
- **El método `clear()` es $O(1)$ para vectores, mientras que para listas es $O(n)$.**
- **Para las restantes funciones el tiempo de ejecución *en el peor caso* es $O(1)$.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con cont. lineales ordenados

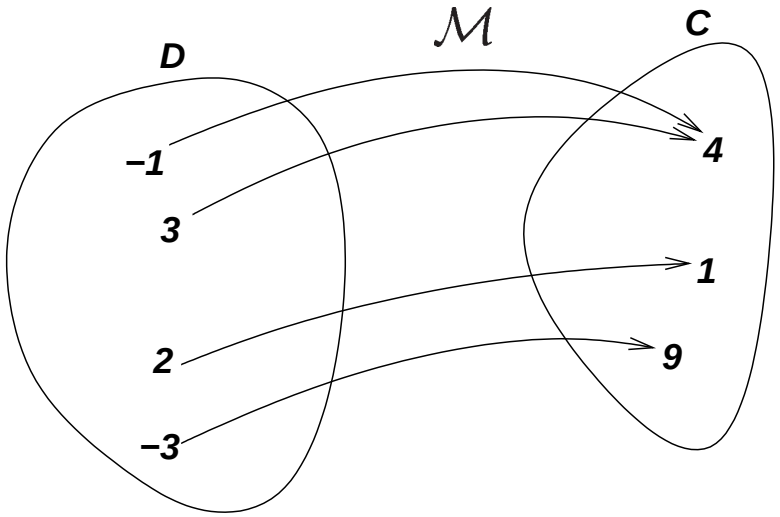
- **Una posibilidad de reducir el tiempo de ejecución es usar contenederos ordenados. Los pares de asignación están ordenados de menor a mayor según la clave.**
- **Si el dominio son los enteros o los reales, entonces se puede usar la relación de orden propia del tipo. Para *"strings"* (cadenas de caracteres) se puede usar el orden *"lexicográfico"*.**
- **Para otros tipos compuestos, para los cuales no existe una relación de orden se pueden definir relaciones de orden *ad-hoc*. En algunos casos estas relaciones de orden no tienen ningún otro interés que ser usadas en este tipo de representaciones o en otros algoritmos relacionados.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas

En el caso de representar la correspondencia mediante una lista ordenada, los pares son ordenados de acuerdo con su campo clave.

$$\mathcal{M} = ((-3, 9), (-1, 4), (2, 1), (3, 4)). \tag{18}$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

- **Ahora `p=find(k)` no debe recorrer toda la lista cuando la clave no está, ya que el algoritmo puede detenerse cuando encuentra una clave *mayor* a la que se busca. Por ejemplo, si hacemos `p=find(0)` en $\mathcal{M}$ podemos dejar de buscar cuando llegamos al par $(2, 1)$.**

$$\mathcal{M} = ((-3, 9), (-1, 4), (2, 1), (3, 4)). \tag{19}$$

- **Sin embargo, al insertar nuevas asignaciones hay que tener en cuenta que no se debe insertar en cualquier posición sino que hay que mantener la lista ordenada. Por ejemplo, si queremos asignar a la clave 0 el valor 7, entonces el par $(0, 7)$ *debe* insertarse entre los pares $(-1, 4)$ y $(2, 1)$, para mantener el orden entre las claves.**

Corresp. con listas ordenadas (cont.)

- Declaración de las clases. `map` contiene una lista `l` que contiene las asignaciones

```
1  class elem_t {
2  private:
3    friend class map;
4    domain_t first;
5    range_t second;
6  };
7  // iterator para map va a ser el mismo que para listas.
8  class map {
9  private:
10   list l;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

- Tanto `p=find(key)` como `p=insert(key,val)` se basan en una función auxiliar `p=lower_bound(key)` que retorna la primera posición donde podría insertarse la nueva clave sin violar la condición de ordenamiento sobre el contenedor. Como casos especiales, si todas las claves son mayores que `key` entonces debe retornar `begin()` y si son todas menores, o la correspondencia está vacía, entonces debe retornar `end()`.

```
1  iterator_t lower_bound(domain_t key) {
2    iterator_t p = l.begin();
3    while (p!=l.end()) {
4      domain_t dom = l.retrieve(p).first;
5      if (dom >= key) return p;
6      p = l.next(p);
7    }
8    return l.end();
9  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

```
1  public:
2    map() { }
3    iterator_t find(domain_t key) {
4      iterator_t p = lower_bound(key);
5      if (p!=l.end() && l.retrieve(p).first == key)
6        return p;
7      else return l.end();
8    }
```

- Si `p` es `end()` o la asignación correspondiente a `p` contiene *exactamente* la clave `key`, entonces debe retornar `p`. Caso contrario, `p` debe corresponder a una posición dereferenciacible, pero cuya clave no es `key` de manera que en este caso no debe retornar `p` sino `end()`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

```
1   iterator_t insert(domain_t key,range_t val) {
2     iterator_t p = lower_bound(key);
3     if (p==l.end() || l.retrieve(p).first != key) {
4       elem_t elem;
5       elem.first = key;
6       p = l.insert(p,elem);
7     }
8     l.retrieve(p).second = val;
9     return p;
10  }
```

- `p=lower_bound(key)` retorna un iterator a la asignación correspondiente a `key` (si `key` tiene un valor asignado) o bien un iterator a la posición donde la asignación a `key` debe ser insertada. En este último caso se inserta un nuevo par . La posición `p` apunta al par correspondiente a `key`, independientemente de si fue creado o no y se reemplaza el valor.
- Notar que `lower_bound()` es declarado `private` en la clase ya que es sólo un algoritmo interno auxiliar para `insert()` y `find()` .

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

```
1    range_t &retrieve(domain_t key) {
2      iterator_t q = find(key);
3      if (q==end()) q=insert(key,range_t());
4      return l.retrieve(q).second;
5    }
```

- El método `val=retrieve(key)` busca una asignación para `key`. Notar que si `key` no está asignado entonces *debe generar una asignación*. El valor correspondiente en ese caso es el que retorna el constructor por defecto (`range_t()`).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con listas ordenadas (cont.)

```
1   bool empty() { return l.begin()==l.end(); }
2   void erase(iterator_t p) { l.erase(p); }
3   int erase(domain_t key) {
4     iterator_t p = find(key); int r = 0;
5     if (p!=end()) { l.erase(p); r = 1; }
6     return r;
7   }
8   iterator_t begin() { return l.begin(); }
9   iterator_t end() { return l.end(); }
10  void clear() { l.erase(l.begin(),l.end()); }
11  int size() { return l.size(); }
12  domain_t key(iterator_t p) {
13    return l.retrieve(p).first;
14  }
15  range_t &value(iterator_t p) {
16    return l.retrieve(p).second;
17  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. Interfase compatible con STL

```
1 template<typename first_t,typename second_t>
2 class pair {
3 public:
4   first_t first;
5   second_t second;
6 };
```

- En vez del tipo `elem_t` se define un template `pair<class first_t,class second_t>`. Este template es usado para `map` y otros contenedores y algoritmos de STL. Los campos `first` y `second` de `pair` son públicos. Esto es un caso muy especial dentro de las STL y la programación orientada a objetos en general ya que en general *se desaconseja permitir el acceso a los campos datos de un objeto*. Otro uso de `pair<>` es para permitir que una función retorne dos valores al mismo tiempo.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. Interfase compatible con STL

```
1 template<typename domain_t,typename range_t>
2 class map {
3 private:
4   typedef pair<domain_t,range_t> pair_t;
5   typedef list<pair_t> list_t;
6   list_t l;
```

- La clase `map` es un template de las clases `domain_t` y `range_t`. Los elementos de la lista serán de tipo `pair<domain_t,range_t>`.
- Para simplificar la escritura de la clase, se definen dos tipos internos `pair_t` y `list_t`. Los elementos de la lista serán de tipo `pair_t`. También se define el tipo público `iterator` que es igual al iterator sobre la lista, pero esta definición de tipo permitirá verlo también externamente como `map<domain_t,range_t>::iterator`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. Interfase compatible con STL (cont.)

```
1  public:
2    typedef typename list_t::iterator iterator;
3    map();
4    iterator find(domain_t key);
5    range_t & operator[](domain_t key);
6    bool empty();
7    void erase(iterator p);
8    int erase(domain_t key);
9    iterator begin();
10   iterator end();
11   void clear();
```

- `val=M.retrieve(key)` se reemplaza sobrecargando el operador `[]`, de manera que con esta interfase la operación anterior se escribe `val=M[key]`. Recordar que tiene el mismo efecto colateral que `retrieve`: *Si `key` no tiene ningún valor asignado, entonces `M[key]` le asigna uno por defecto.* Además, igual que `retrieve`, `M[key]` retorna una referencia de manera que es válido usarlo como miembro izquierdo, como en `M[key]=val`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. Interfase compatible con STL (cont.)

```
1  map<int,double> sueldo;
2  while(1) {
3    cout << "Ingrese nro. documento > ";
4    int doc;
5    double salario;
6    cin >> doc;
7    if (!doc) break;
8    map<int,double>::iterator q = sueldo.find(doc);
9    if (q==sueldo.end()) {
10     cout << "Ingrese salario mensual: ";
11     cin >> salario;
12     sueldo[doc]=salario;
13   } else {
14     cout << "Doc: " << doc << ", salario: "
15          << sueldo[doc] << endl;
16   }
17 }
18 cout << "No se ingresan mas sueldos..." << endl;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. Interfase compatible con STL (cont.)

(Notación: mejor/promedio/peor)

Operación	lista	vector
`find(key)`	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
`M[key]` (no existente)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`M[key]` (existente)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
`erase(key)`	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`key`, `value`, `begin`, `end`,	$O(1)$	$O(1)$
`clear`	$O(n)$	$O(1)$

Corresp. con vectores ordenados

- **La ganancia real de usar contenedores ordenados es en el caso de vectores ya que en ese caso podemos usar el algoritmo de *"búsqueda binaria"* (*"binary search"*) en cuyo caso `p=lower_bound(k)` resulta ser $O(\log n)$ *en el peor caso*.**
- **El código se basa en la clase `vector` de STL, pero en realidad con modificaciones menores se podrían reemplazar los vectores de STL por arreglos estándar de C.**
- **La correspondencia almacena las asignaciones (de tipo `pair_t`) en un `vector<pair_t> v`. Para el tipo `map<>::iterator` usamos directamente el tipo entero.**

Corresp. con vectores ordenados (cont.)

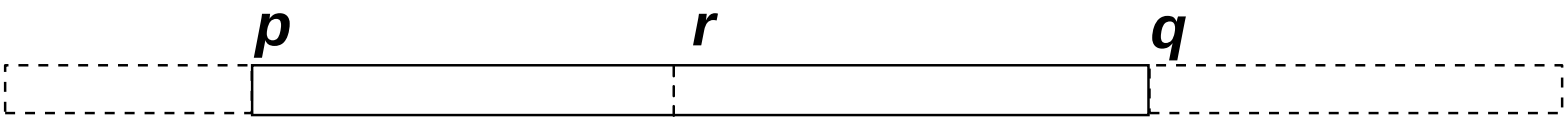
- **`lower_bound()`. El algoritmo se basa en ir refinando un rango $[p, q)$ tal que la clave buscada esté garantizado siempre en el rango, es decir $k_p \leq k < k_q$, donde k_p, k_q son las claves en las posiciones p y q respectivamente y k es la clave buscada.**
- **Primero se verifican una serie de casos especiales hasta obtener un rango válido. Si el vector está vacío, entonces `lower_bound` retorna 0 ya que entonces debe insertar en `end()` que en ese caso vale 0. Lo mismo si $k < k_0$, ya que en ese caso la clave va en la primera posición. Si ninguno de estos casos se aplica, entonces el rango $[p, q)$, con $p = 0$ y $m =$`v.size()` es un rango válido, es decir $k_p \leq k < k_q$.**

Corresp. con vectores ordenados (cont.)

- **Una vez que tenemos un rango válido $[p, q)$ podemos *refinarlo* haciendo**

$$r = \text{floor}((p + q)/2) \tag{20}$$

Esto divide $[p, q)$ en dos subrangos disjuntos $[p, r)$ y $[r, q)$. y comparando la clave *k* con la que está almacenado en la posición r, k_r. Si $k \geq k_r$ entonces el nuevo rango es el $[r, q)$, mientras que si no es el $[p, r)$.



Corresp. con vectores ordenados (cont.)

- **Notar que si el rango $[p, q)$ es de longitud par, esto es $n = q - p$ es par, entonces los dos nuevos rangos son iguales, de longitud igual a la mitad $n = (q - p)/2$. Si el número de elementos es inicialmente una potencia de 2, digamos $n = 2^m$, entonces después del primer refinamiento la longitud será 2^{m-1}, después de dos refinamientos 2^{m-2}, hasta que, después de m refinamientos la longitud se reduce a $2^0 = 1$ y el algoritmo se detiene. Entonces $T(n) = m = \log_2 n$. Si n no es una potencia de dos, entonces el número de refinamientos es $m = \text{floor}(\log_2 n) + 1$.**
- **Mucho mejor que el $O(n)$ que teníamos con las listas.**
- **Búsqueda binaria no se puede aplicar a listas ya que estas no son contenedores de *"acceso aleatorio"*. Para vectores, acceder al elemento j-ésimo es una operación $O(1)$, mientras que para listas involucra recorrer toda la lista es $O(j)$.**

Corresp. con vectores ordenados (cont.)

- La implementación para vectores es muy similar a la de listas. El único cambio significativo es `lower_bound`.
- En vez de la lista `l` tenemos un `vector<T> v`

```
1   iterator lower_bound(domain_t key) {
2     int p=0, q=v.size(), r;
3     if (!q || v[p].first >key) return 0;
4     while (q-p > 1) {
5       r = (p+q)/2;
6       range_t kr = v[r].first;
7       if (key > kr) p=r;
8       else if (key < kr) q=r;
9       else if (kr==key) return r;
10    }
11    if (v[p].first == key) return p;
12    else return q;
13  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Corresp. con vectores ordenados (cont.)

(Notación: mejor/promedio/peor)

Operación	lista	vector
`find(key)`	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
`M[key]` (no existente)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
`M[key]` (existente)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
`erase(key)`	$O(1)$	$O(1)/O(n)/O(n)$
`key`, `value`, `begin`, `end`,	$O(1)$	$O(1)$
`clear`	$O(n)$	$O(1)$

Arboles

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles

- **Los árboles permiten organizar un conjunto de objetos en forma *jerárquica*.**
- **Ejemplos típicos:**
  - ▷ ***Diagrama de organización* de las empresas o instituciones**
  - ▷ **La estructura de un *sistema de archivos* en una computadora.**
  - ▷ **Representar *fórmulas*.**
- **En general para representar grandes sistemas en sistemas más pequeños en forma *recursiva***
- **Permiten acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos. Por ejemplo, en mi cuenta poseo unos *61,000 archivos* organizados en unos *3500 directorios* a los cuales puedo acceder con un máximo de 10 cambios de directorio (*en promedio unos 5*).**

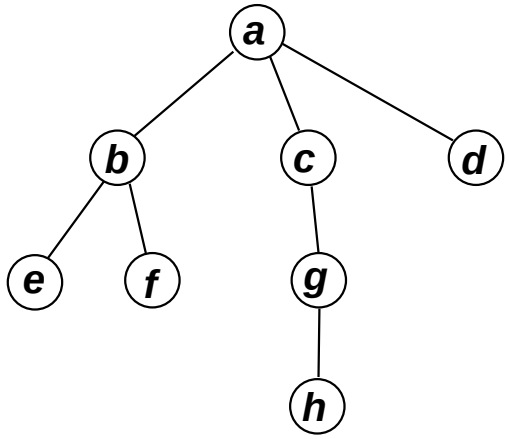
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles (cont.)

Sorprendentemente *no existe un contenedor STL de tipo árbol*, si bien varios de los otros contenedores (como conjuntos y correspondencias) están implementados internamente en términos de árboles. Esto se debe a que en la filosofía de las STL el árbol es considerado o bien como un *subtipo del grafo* o bien como una entidad demasiado básica para ser utilizada directamente por los usuarios.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Nomenclatura básica de árboles

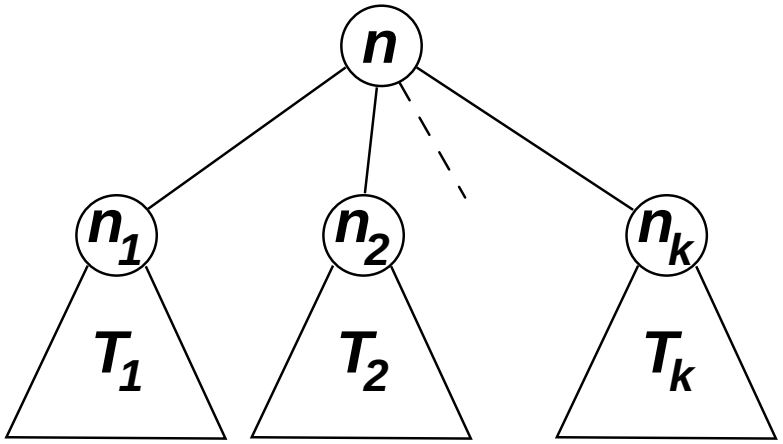


- **Un árbol es una colección de *"nodos"*, uno de los cuales es la *"raíz"*.**
- **Existe una relación de parentesco por la cual cada nodo tiene un y sólo un *"padre"*, salvo la raíz que no lo tiene.**
- **El nodo es el concepto análogo al de *"posición"* en la lista**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

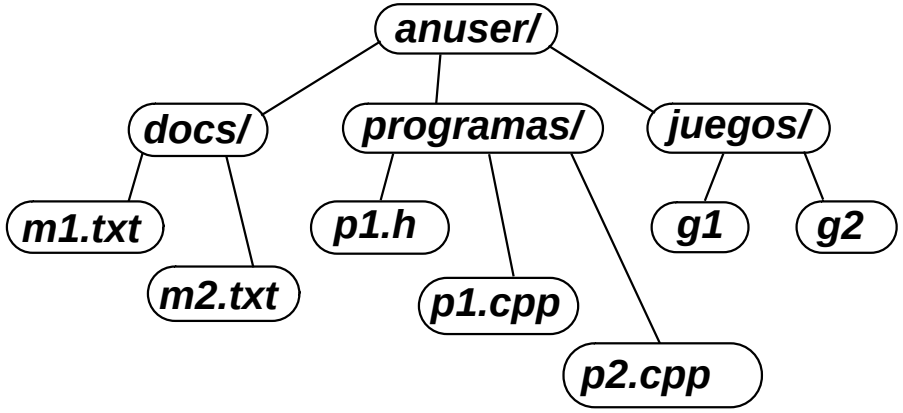
Nomenclatura básica de árboles (cont.)

- **Definición recursiva de un árbol:**
  - ▷ **Un nodo sólo es un árbol**
  - ▷ **Si n es un nodo y T_1, T_2, ..., T_k son árboles con raíces n_1, ..., n_k entonces podemos construir un nuevo árbol que tiene a n como raíz y donde n_1, ..., n_k son *"hijos"* de n.**



- **También es conveniente postular la existencia de un *"árbol vacío"* que llamaremos Λ.**

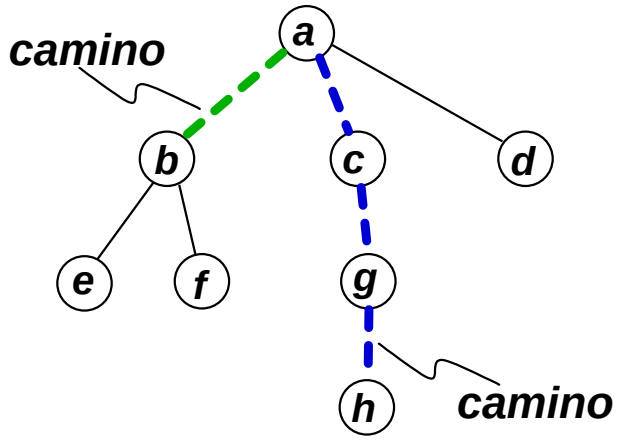
Nomenclatura básica de árboles (cont.)



- **Arbol que representa los archivos en un sistema de archivos.**
- **`anuser/` contiene 3 subdirectorios `docs/`, `programas/` y `juegos/`, los cuales a su vez contienen una serie de archivos.**
- **Relación entre nodos hijos y padres corresponde a pertenencia: a es hijo b, si el archivo a *pertenece* al directorio b.**
- **En otras aplicaciones la relación padre/hijo puede representar otra cosa.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

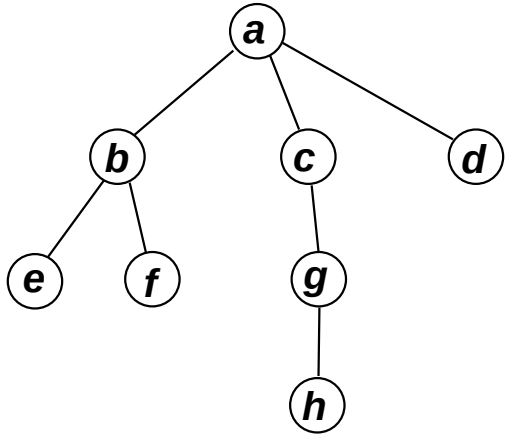
Camino en un árbol



- $n_1, n_2, \ldots, n_k$ **secuencia de nodos tales que** n_i **es padre de** n_{i+1} **para** $i = 1 \ldots k-1$**, entonces decimos que esta secuencia de nodos es un** ***"camino"*** **(*"path"*)**
- **La** ***"longitud"*** **de un camino es igual al número de nodos en el camino menos uno.**
- **Ejemplos** (a, b) **de longitud 1,** (a, c, g, h) **de longitud 3.**
- **Siempre existe un camino de longitud 0 de un nodo a sí mismo.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

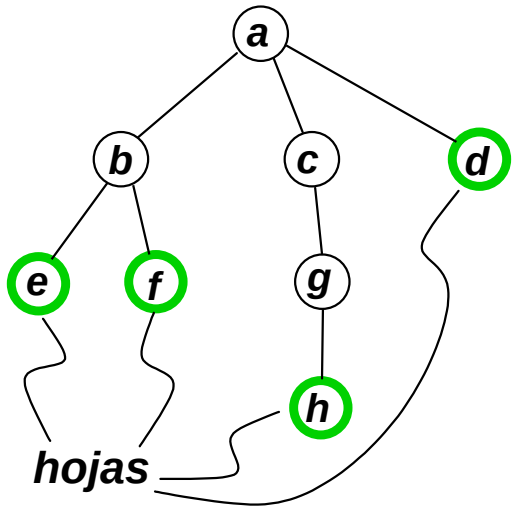
Descendientes y antecesores.



- **Si existe un camino que va del nodo a al b entonces decimos que a es antecesor de b y b es descendiente de a.**
- **Un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0.**
- **Decimos que a es *descendiente (antecesor) propio* de b si a es descendiente (antecesor) de b, *pero* $a \neq b$.**
- **En el ejemplo de la figura a es antecesor propio de c, f y d,**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

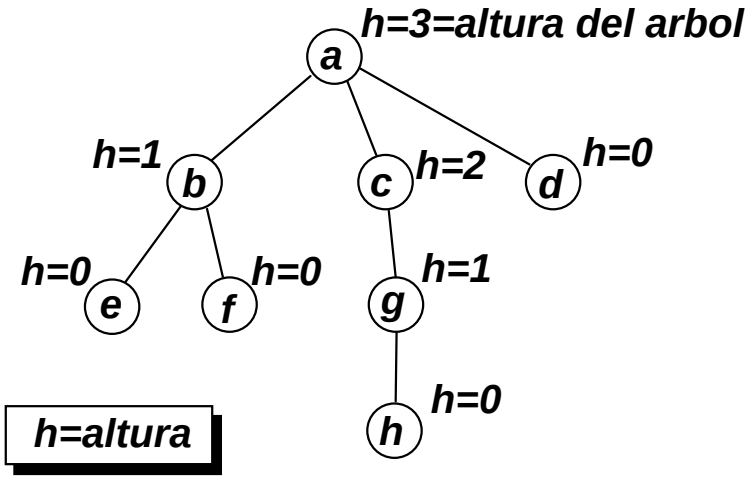
Hojas



- **Un nodo que no tiene hijos es una *"hoja"* del árbol. (Recordemos que, por contraposición el nodo que no tiene padre es único y es la raíz.) En el ejemplo, los nodos e, f, h y d son hojas.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Altura de un nodo.

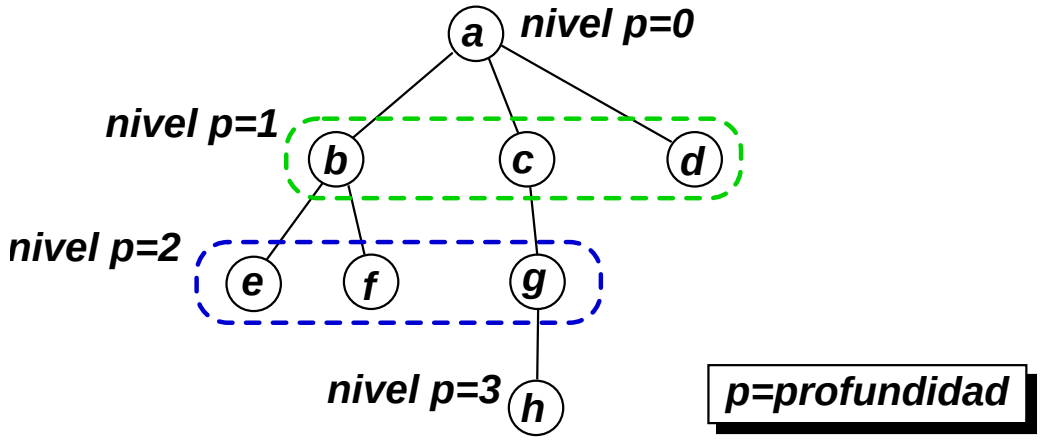


La *altura de un nodo* en un árbol es la *máxima longitud de un camino que va desde el nodo a una hoja*. Por ejemplo, el árbol de la figura la altura del nodo c es 2. La altura del árbol es la altura de la raíz. La altura del árbol del ejemplo es 3. Notar que, para cualquier nodo n

$$\textbf{altura}(n) = \begin{cases} 0 & ; \textbf{ si } n \textbf{ es una hoja} \\ 1 + \max_{s=\textbf{hijo de } n} \textbf{altura}(s) & ; \quad \textbf{si no lo es.} \end{cases}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

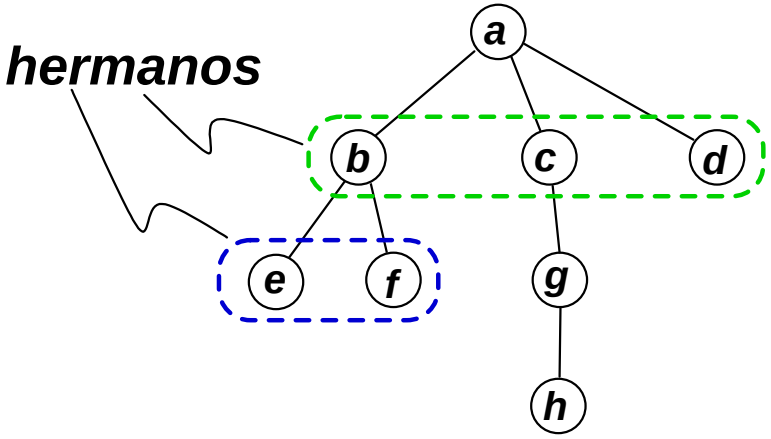
Profundidad de un nodo. Nivel.



- La *"profundidad"* o de un nodo es la *longitud del único camino que va desde el nodo a la raíz.*
- Un *"nivel"* en el árbol es el conjunto de todos los nodos que están a una misma profundidad.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Nodos hermanos



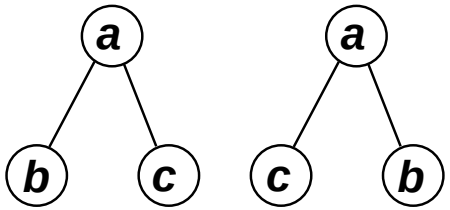
- **Se dice que los nodos que tienen un mismo padre son *"hermanos"* entre sí.**
- **Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles ordenados orientados

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

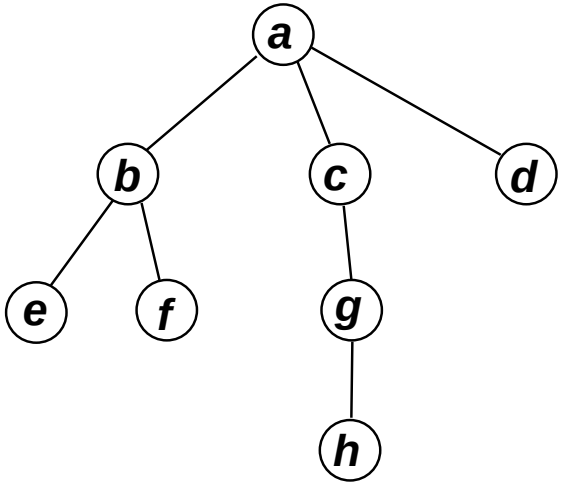
Orden de los nodos



- **En este capítulo, estudiamos árboles para los cuales el *orden* entre los hermanos es relevante.**
- **Es decir, los árboles de la figura *son diferentes* ya que si bien a tiene los mismos hijos, están en diferente orden.**
- **Por eso a estos árboles se les llama *"árboles ordenados orientados"* (AOO)**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

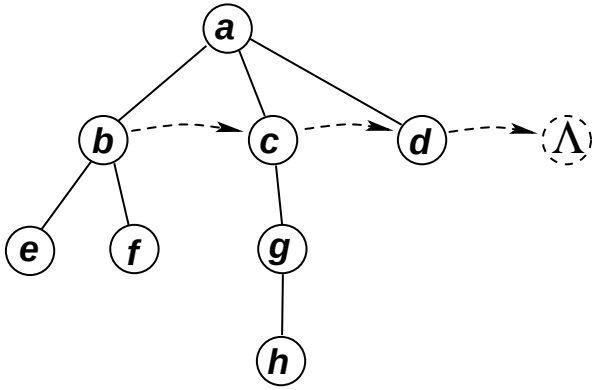
Orden de los nodos (cont.)



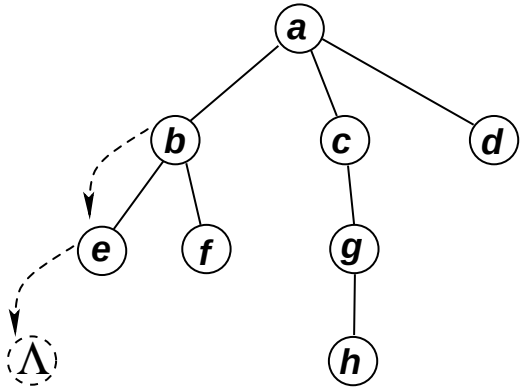
- **Decimos que el nodo c está a la derecha de b, o también que c es es el hermano derecho de b.**
- **También decimos que b es el *"hijo más a la izquierda"* de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes de c y b, respectivamente.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden de los nodos (cont.)



avanza por hermano derecho



avanza por hijo mas izquierdo

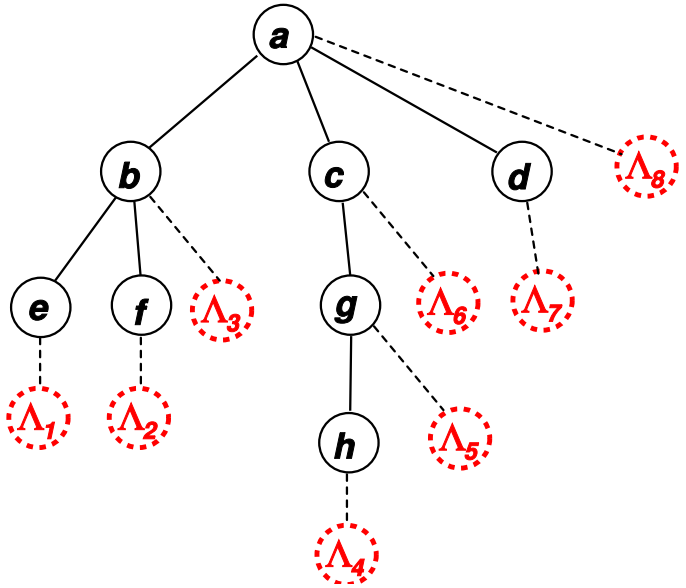
Podemos pensar al árbol como una *lista bidimensional*. Así como en las listas se puede avanzar *linealmente* desde el comienzo hacia el fin, en cada nodo del árbol podemos avanzar en dos direcciones

- **Por el *hermano derecho*, de esta forma se recorre toda la lista de hermanos de izquierda a derecha.**
- **Por el *hijo más izquierdo*, tratando de descender lo más posible en profundidad.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Posiciones no dereferenciables

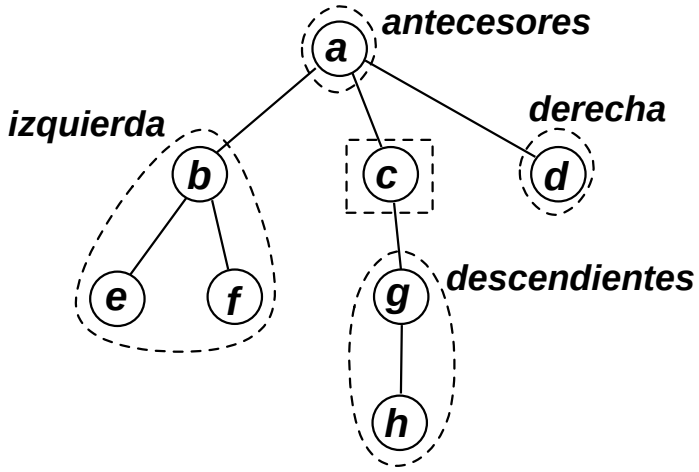
- **Avanzando por el hermano derecho el recorrido termina en el último hermano a la derecha. Por analogía con la posición `end()` en las listas, asumiremos que después del último hermano existe un *nodo ficticio no dereferenciable*.**

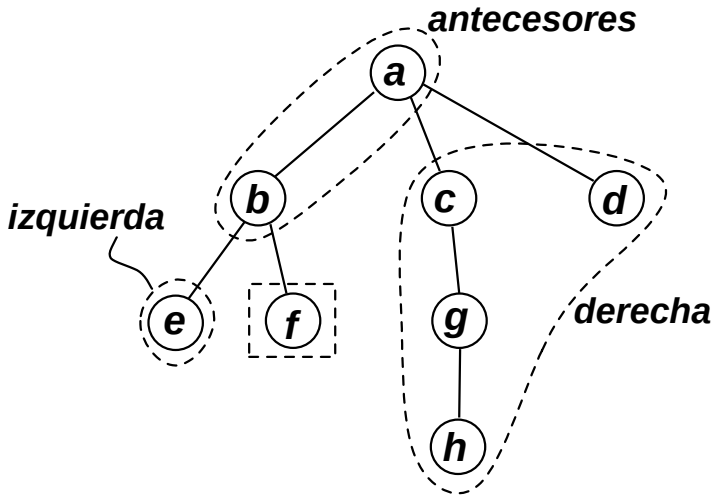


- **También asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable.**
- **Notar que, a diferencia de la lista donde hay una sola posición no dereferenciable, en un árbol *puede haber muchas*, las cuales simbolizaremos con Λ cuando dibujamos el árbol.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Particionamiento del conjunto de nodos





Dados dos nodos cualquiera m y n una y sólo una de las siguientes afirmaciones es cierta

- $m = n$
- m **es** ***antecesor*** **propio de** n
- m **es** ***descendiente*** **propio de** n
- m ***está a la derecha*** **de** n
- m ***está a la izquierda*** **de** n

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

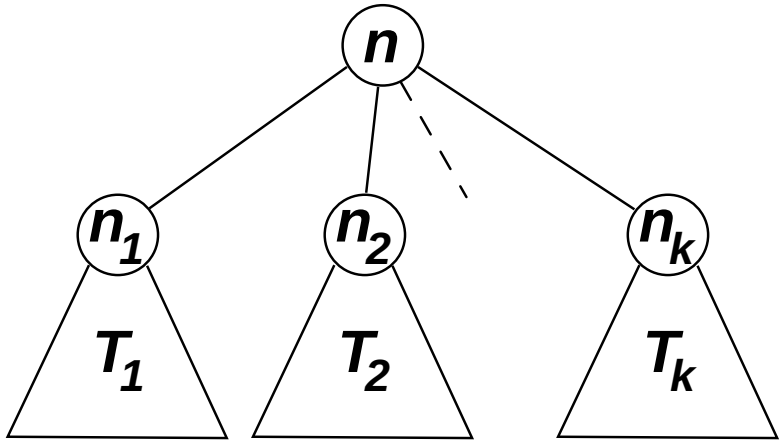
Particionamiento del conjunto de nodos (cont.)

Dicho de otra manera, dado un nodo n el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos *disjuntos* a saber

$$N = \{n\} \cup \{\textbf{descendientes-propios}(n)\} \cup \{\textbf{antecesores-propios}(n)\}$$
$$\cup \{\textbf{derecha}(n)\} \cup \{\textbf{izquierda}(n)\}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden previo



Dado un nodo n con hijos $n_1, n_2, \ldots n_m$, el *"listado en orden previo"* (*"preorder"*) del nodo n se define como

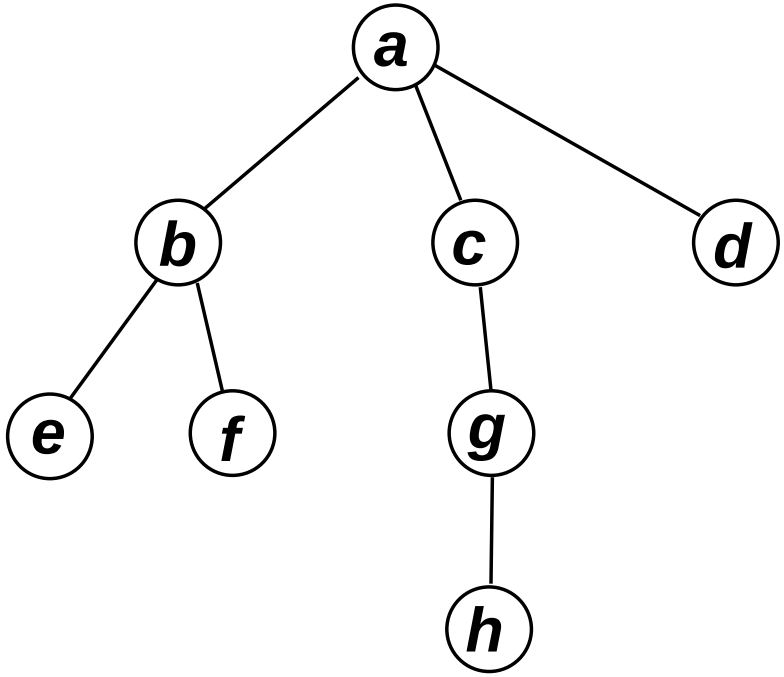
$$\text{oprev}(\Lambda) = <\textbf{lista vacía}>$$

$$\text{oprev}(n) = n, \text{oprev}(n_1), \text{oprev}(n_2), \ldots, \text{oprev}(n_m)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden previo (cont.)

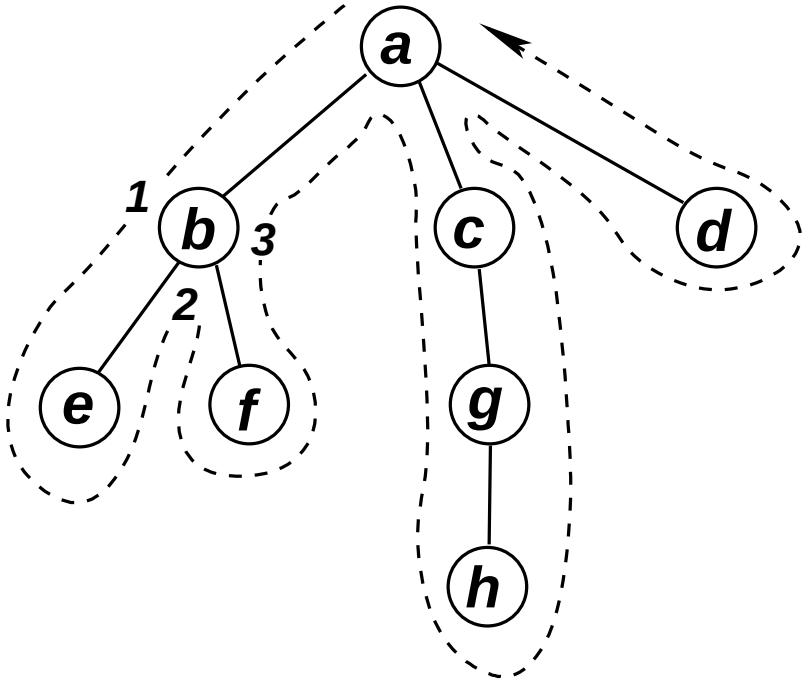
$$\begin{aligned}
\text{oprev}(a) &= a, \text{oprev}(b), \text{oprev}(c), \text{oprev}(d) \\
&= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\
&= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\
&= a, b, e, f, c, g, \text{oprev}(h), d \\
&= a, b, e, f, c, g, h, d
\end{aligned}$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden previo (cont.)

$$\text{oprev}(a) = (a, b, e, f, c, g, h, d)$$

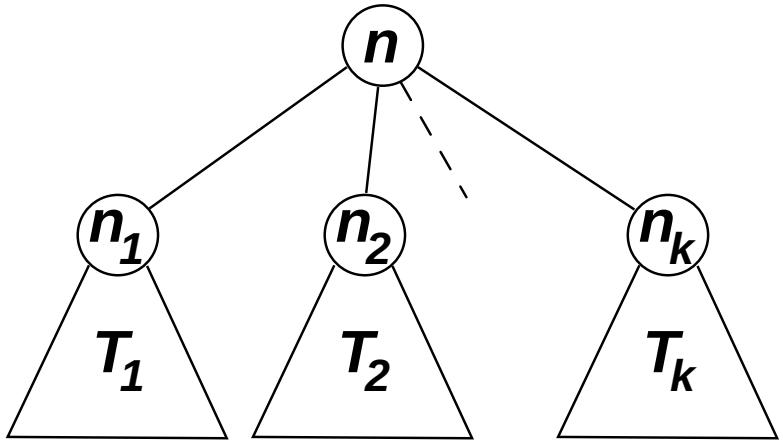


- **Recorremos el borde del árbol en el sentido contrario a las agujas del reloj**
- **Dado un nodo como el b el camino pasa cerca de él en varios puntos.**

- **El orden previo consiste en *listar los nodos una sola vez, la primera vez que el camino pasa cerca del árbol.***

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior



El *"orden posterior"* (*"postorder"*) se puede definir en forma análoga al orden previo

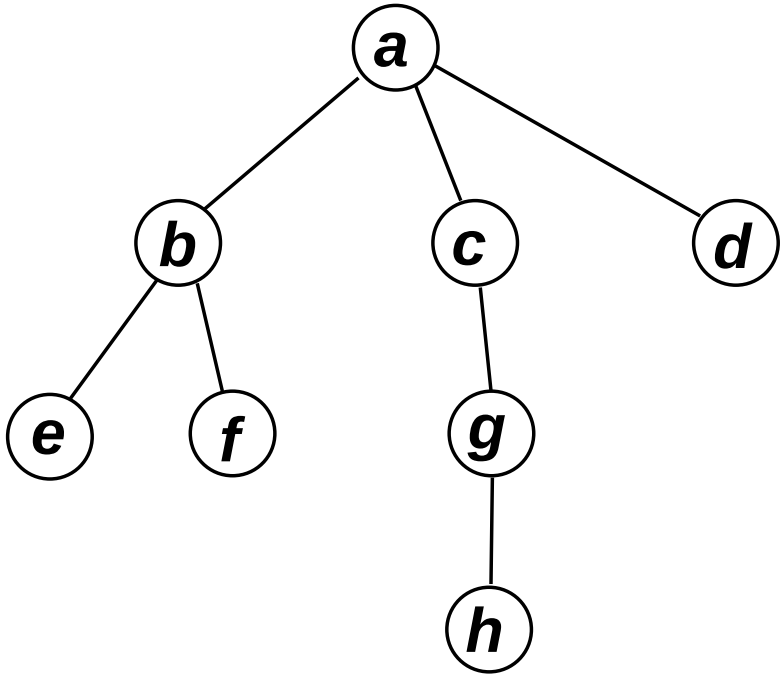
$$\mathrm{opost}(\Lambda) = <\textbf{lista vacía}>$$

$$\mathrm{opost}(n) = \mathrm{opost}(n_1), \mathrm{opost}(n_2), \ldots, \mathrm{opost}(n_m), n$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

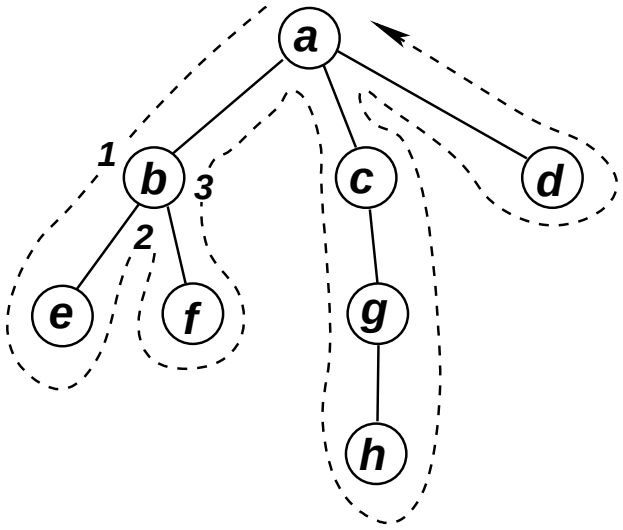
Orden posterior (cont.)

$$\begin{aligned}\text{opost}(a) &= \text{opost}(b), \text{opost}(c), \text{opost}(d), a \\ &= \text{opost}(e), \text{opost}(f), b, \text{opost}(g), c, d, a \\ &= e, f, b, \text{opost}(h), g, c, d, a \\ &= e, f, b, h, g, c, d, a\end{aligned}$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior (cont.)

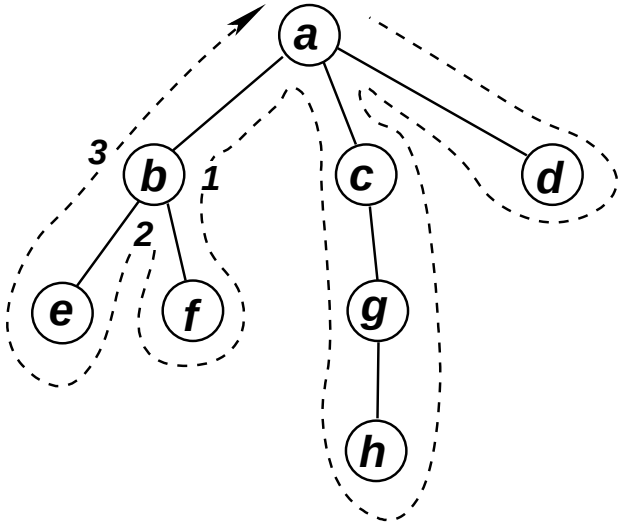


$$\mathrm{opost}(a) = (e, f, b, h, g, c, d, a)$$

- **Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo *la última vez que el recorrido pasa por al lado del mismo*.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior (cont.)



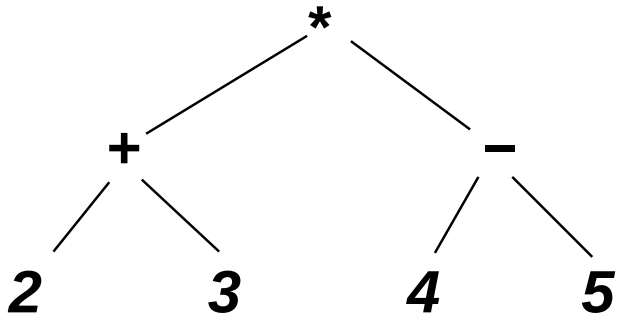
- **Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la *primera vez* que el camino pasa cera de ellos. Una vez que la lista es obtenida, *invertimos la lista*. En el caso de la figura el recorrido en sentido contrario daría (a, d, c, g, h, b, f, e). Al invertirlo queda (e, f, b, h, g, c, d, a)**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden simétrico

Existe otro orden que se llama *"simétrico"*, pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

Orden posterior y la notación polaca invertida



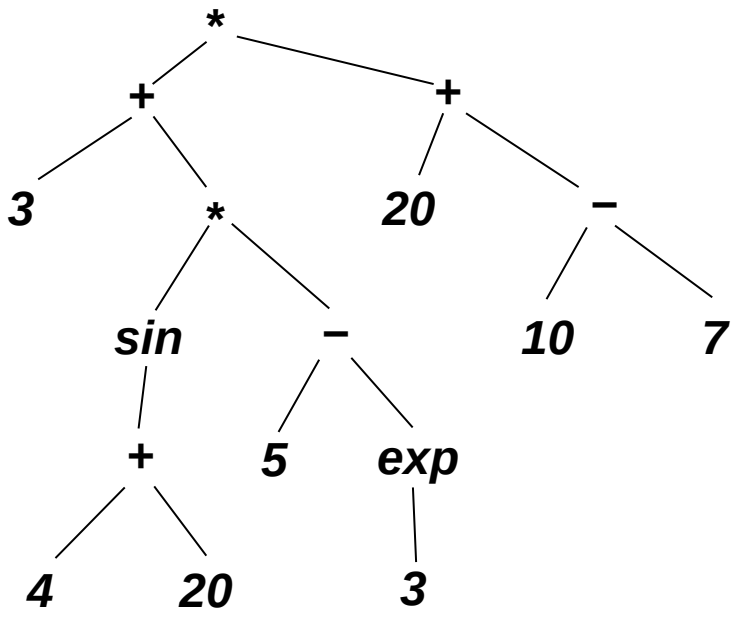
Las *expresiones matemáticas* como $(2+3)*(4-5)$ se pueden poner en forma de árbol como se muestra en la figura.

- **Para *operadores binarios* de la forma $a+b$ se pone el operador $(+)$ como padre de los dos operandos (a y b). Los operandos pueden ser a su vez expresiones. *Funciones binarias* como $rem(10,5)$ (rem es la función resto) se tratan de esta misma forma.**
- ***Operadores unarios* (como -3) y funciones (como $\sin(20)$) se escriben poniendo el operando como hijo del operador o función.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior y la notación polaca invertida (cont.)

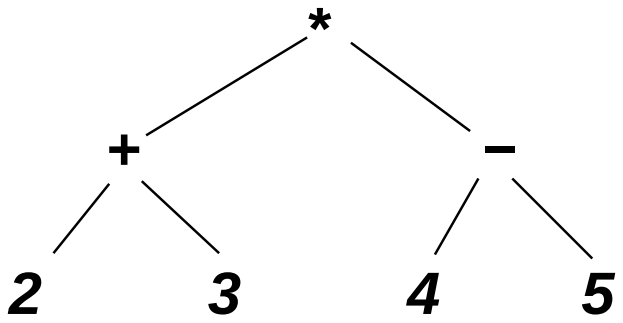
- ***Operadores asociativos*** **con más de dos operandos (como $1 + 3 + 4 + 9$) deben asociarse de a 2 (como en $(((1 + 3) + 4) + 9)$).**



$$(3 + \sin(4 + 20) * (5 - \exp(3))) * (20 + 10 - 7)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior y la notación polaca invertida (cont.)



El listado en orden posterior de árboles de expresiones coincide con la *notación polaca invertida (RPN)*.

$$\text{rpn} = (2,\ 3,\ +,\ 4,\ 5,\ -,\ *)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una calculadora RPN con una pila

La forma más simple de implementar una *calculadora* es usando una *pila* a condición que el usuario ingrese los operandos y operadores en *notación RPN*.

- **Si el usuario ingresó un *operando*, entonces simplemente se almacena en la pila.**
- **Si ingresó un *operador binario* θ se extraen dos operandos del tope de la pila, digamos t y u y se inserta $u\,\theta\,t$ en el tope de la pila.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una calculadora RPN con una pila (cont.)

Para la expresión $(2+3)*(4-5)$**, tenemos** $\mathrm{rpn} = (2,\ 3,\ +,\ 4,\ 5,\ -,\ *)$**.**

Ingresa	Tipo	Pila
2	operando	2
3	operando	3,2
+	operador	5
4	operando	4,5
5	operando	5,4,5
-	operador	-1,5
*	operador	-5

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Una calculadora RPN con una pila (cont.)

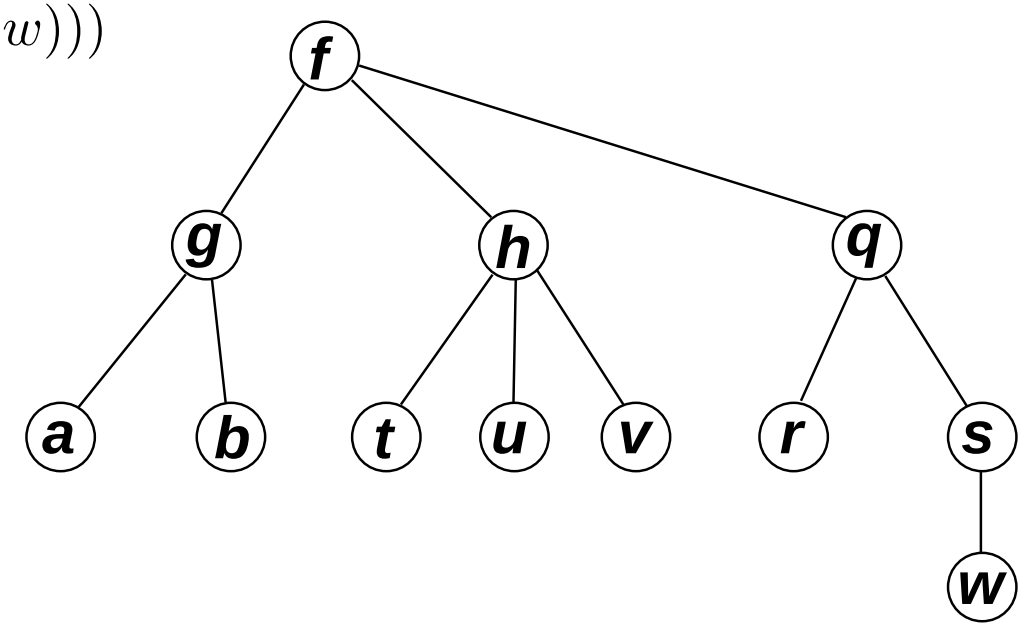
- **El algoritmo puede extenderse fácilmente a *funciones con un número arbitrario de variables* (como `exp()`, `cos()` ...), sólo que en ese caso se extrae el número de elementos apropiados de la pila, se le aplica el valor y el resultado es introducido en la misma.**
- **Notar que en general debe *invertirse* el orden de los argumentos al sacarlos de la pila.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp para árboles

Una *expresión matemática compleja* que involucra funciones cuyos argumentos son a su vez llamadas a otras funciones *puede ponerse en forma de árbol.* En este caso cada función es un *nodo* cuyos hijos son los *argumentos* de la función. Por ejemplo,

$$f(g(a,b),h(t,u,v),q(r,s(w)))$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp para árboles (cont.)

En el lenguaje *Lisp* la llamada a una función $f(x, y, z)$ se escribe de la forma `(f x y z)`, de manera que

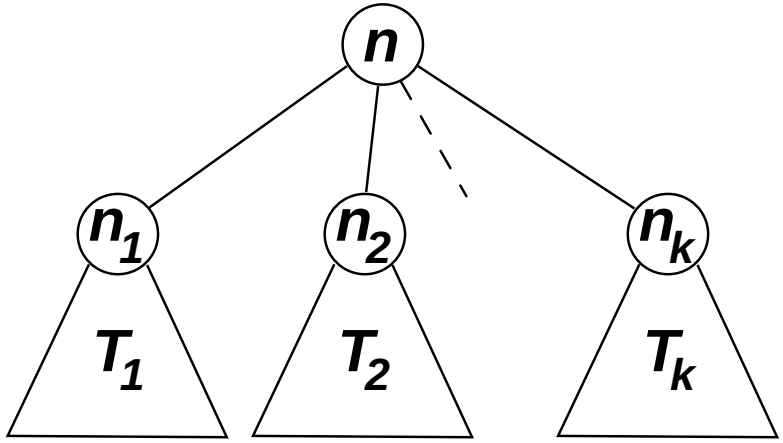
$$f(g(a, b), h(t, u, v), q(r, s(w)))$$

se escribiría como

```
1   (f (g a b) (h t u v) (q r (s w)))
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp para árboles (cont.)



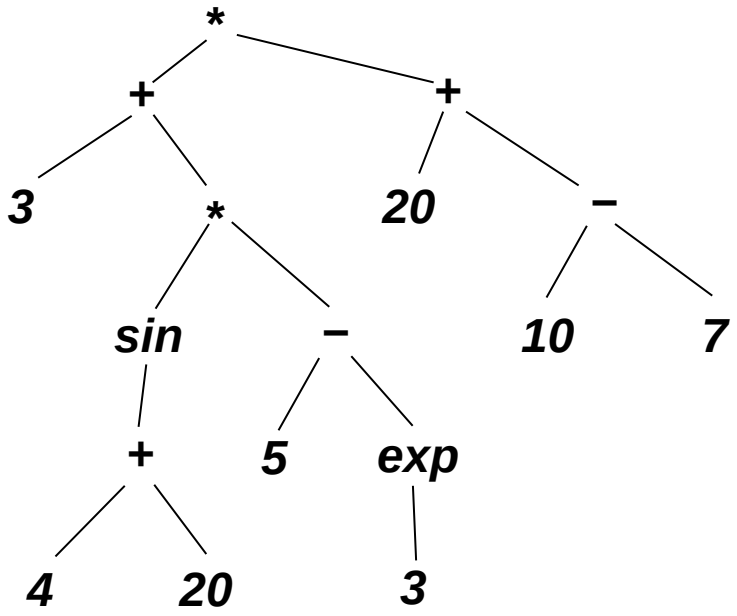
Notemos que el orden de los nodos *es igual al del orden previo*. Se puede dar una definición precisa de la notación Lisp como para el caso de los órdenes previo y posterior:

$$\text{lisp}(n) = \begin{cases} \textbf{si } n \textbf{ es una hoja:} & n \\ \textbf{caso contrario:} & (n\ \text{lisp}(n_1)\ \text{lisp}(n_2)\ \ldots\ \text{lisp}(n_m)) \end{cases}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp para árboles (cont.)

Para expresiones más complejas, la forma Lisp para el árbol da el código Lisp correspondiente




```
1   (* (+ 3 (* (sin (+ 4 20)) (- 5 (exp 3)))) (+ 20 (- 10 7)))
```

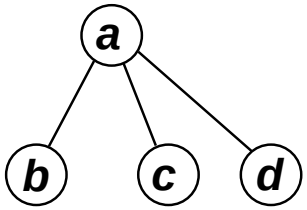

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

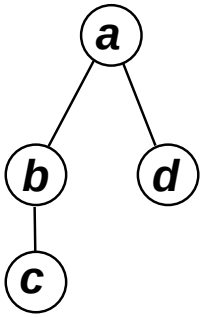
Notación Lisp para árboles (cont.)

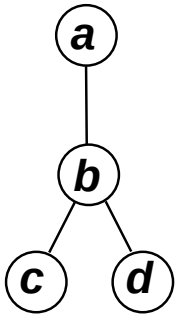
- **Es evidente que existe una *relación unívoca* entre un árbol y su notación Lisp.**
- **Los paréntesis dan la *estructura adicional* que permite establecer la relación unívoca.**
- **La utilidad de esta notación es que permite fácilmente *escribir árboles en una línea de texto,* sin tener que recurrir a un gráfico. Basado en esta notación, es fácil escribir una función que convierta *un árbol a una lista y viceversa*.**
- **También permite *"serializar"* un árbol, es decir, convertir una estructura *"bidimensional"* como es el árbol, en una estructura unidimensional como es una lista.**
- **La utilidad de serializar estructuras complejas es que después es más simple realizar operaciones como *guardar* las estructuras a disco para después leerlas, *enviarlas* a través de la red o via mensajes entre diferentes procesos.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reconstrucción del árbol



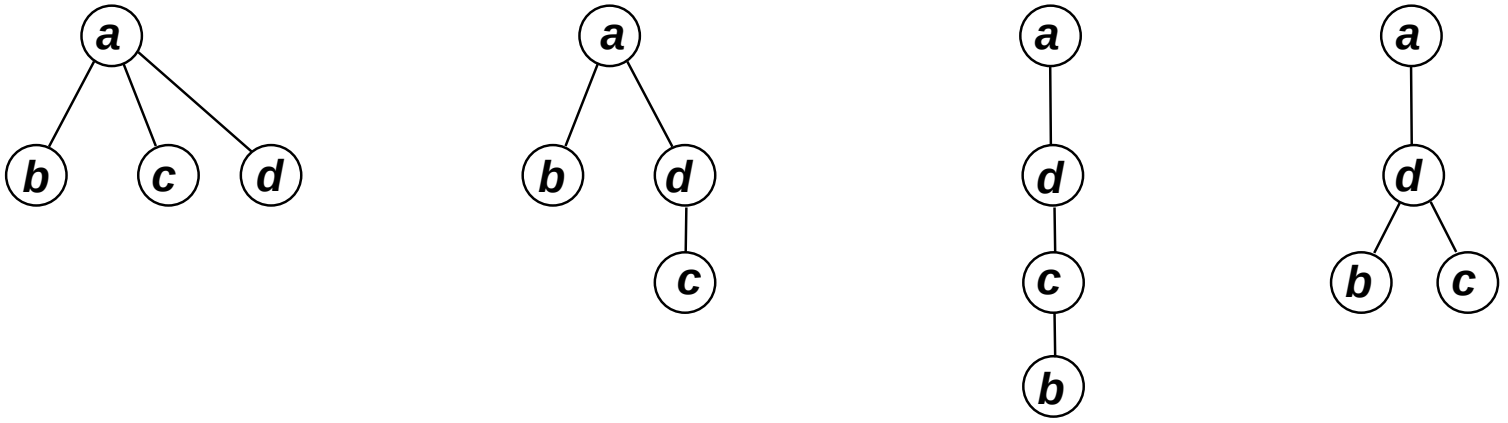




Podemos preguntarnos si podemos *reconstruir un árbol* a partir de su listado en orden previo. Si tal cosa fuera posible, entonces sería fácil representar árboles en una computadora, almacenando dicha lista. Sin embargo puede verse fácilmente que *árboles distintos pueden dar el mismo orden previo.*
Todos estos árboles tienen orden previo (a, b, c, d)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reconstrucción del árbol (cont.)

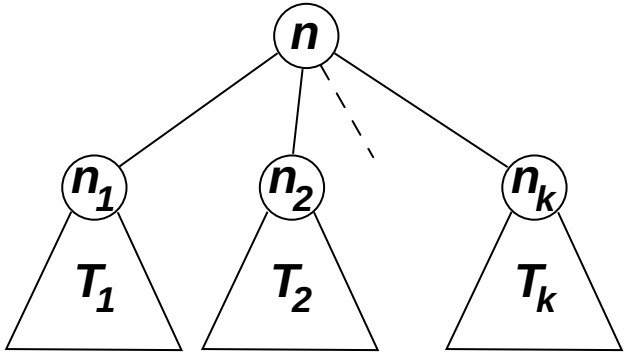


Lo mismo ocurre con el orden posterior. Todos estos árboles tienen orden posterior (b, c, d, a)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reconstrucción del árbol (cont.)

Sin embargo, es destacable que, *dado el orden previo y posterior de un árbol sí se puede reconstruir el árbol*. Primero notemos que implica que el orden de los nodos queda así



$$\text{oprev}(n) = (n, n_1, \textbf{descendientes}(n_1), n_2, \textbf{descendientes}(n_2), \ldots, n_k, \textbf{descendientes}(n_k))$$

mientras que

$$\text{opost}(n) = (\textbf{descendientes}(n_1), n_1, \textbf{descendientes}(n_2), n_2, \ldots, \textbf{descendientes}(n_k), n_k, n).$$

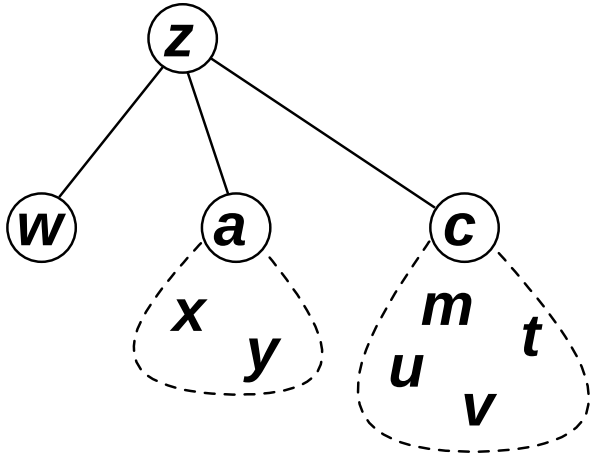
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reconstrucción del árbol (cont.)

Consigna: **Encontrar el árbol A tal que**

$$\textbf{orden previo} = (z, w, a, x, y, c, m, t, u, v)$$

$$\textbf{orden posterior} = (w, x, y, a, t, u, v, m, c, z)$$



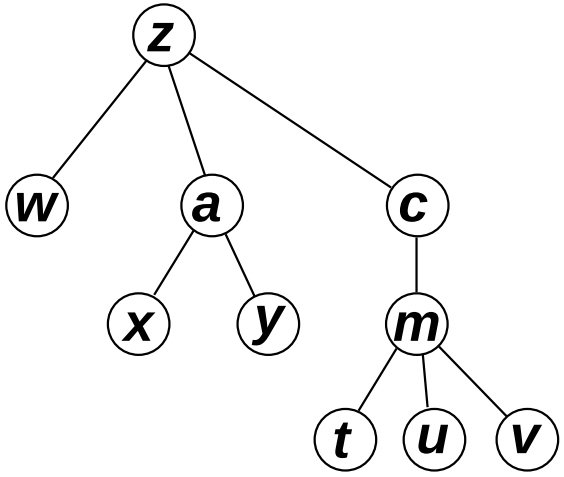
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reconstrucción del árbol (cont.)

Ahora bien, para hallar la estructura de los descendientes de c vemos que

$$\text{oprev}(c) = (c, m, t, u, v)$$

$$\text{opost}(c) = (t, u, v, m, c)$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmos para listar nodos. Orden previo

Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su *naturaleza intrínsecamente recursiva.*

$$\text{oprev}(n) = (n, \text{oprev}(n_1), \text{oprev}(n_2), \ldots, \text{oprev}(n_m))$$

```
1 void preorder(tree &T,iterator n,list &L) {
2   L.insert(L.end(),/* valor en el nodo 'n'... */);
3   iterator c = /* hijo mas izquierdo de n... */;
4   while (/* 'c' no es 'Lambda'... */) {
5     preorder(T,c,L);
6     c = /* hermano a la derecha de c... */;
7   }
8 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Orden posterior

$$\text{opost}(n) = (\text{opost}(n_1), \text{opost}(n_2), \ldots, \text{opost}(n_m), n)$$

```
1 void postorder(tree &T,iterator n,list &L) {
2   iterator c = /* hijo mas izquierdo de n . . . */;
3   while (c != T.end()) {
4     postorder(T,c,L);
5     c = /* hermano a la derecha de c . . . */;
6   }
7   L.insert(L.end(),/* valor en el nodo 'n' . . . */);
8 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp

$$\text{lisp}(n) = \begin{cases} \textbf{si } n \textbf{ es una hoja:} & n \\ \textbf{caso contrario:} & (n\ \text{lisp}(n_1)\ \text{lisp}(n_2)\ \ldots\ \text{lisp}(n_m)) \end{cases}$$

```
1  void lisp_print(tree &T,iterator n) {
2    iterator c = /* hijo mas izquierdo de n . . . */;
3    if (/* 'c' es 'Lambda'. . . */) {
4      cout << /* valor en el nodo 'n'. . . */;
5    } else {
6      cout << "(" << /* valor de 'n' . . . */;
7      while (/* 'c' no es 'Lambda'. . . */) {
8        cout << " ";
9        lisp_print(T,c);
10       c = /* hermano derecho de c . . . */;
11     }
12     cout << ")";
13   }
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Inserción en árboles

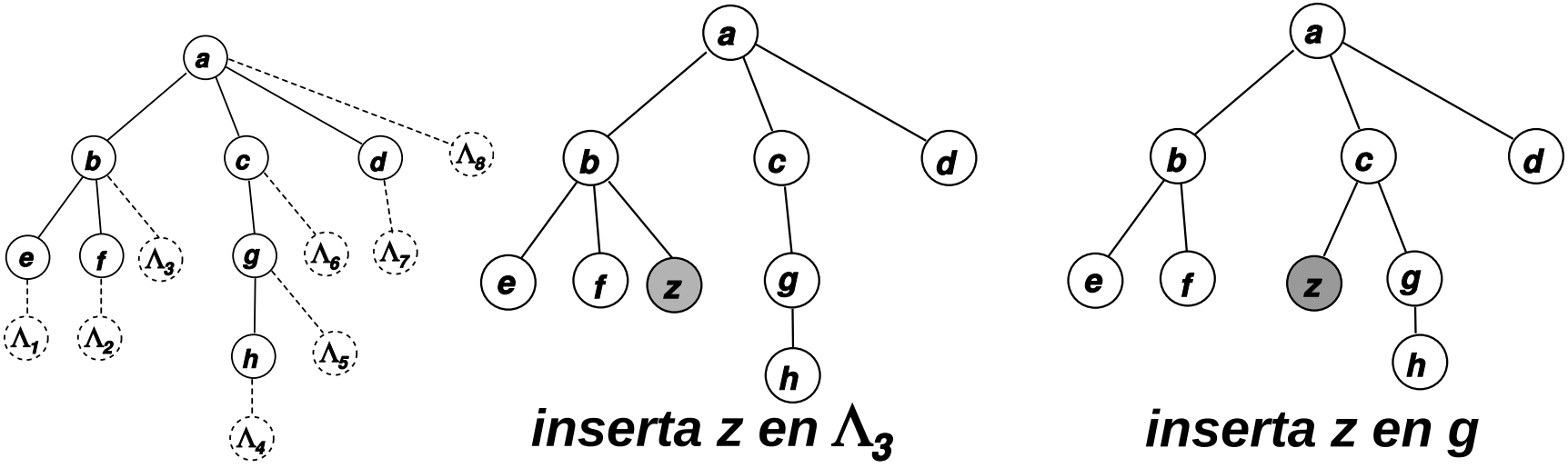
Como en las listas, las *operaciones de inserción* toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

- **Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a *generar un nuevo nodo* en donde estaba el nodo ficticio Λ.**
- **Cuando insertamos un nodo en una posición *dereferenciable*, entonces simplemente el elemento pasa a *generar un nuevo nodo hoja* en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos.**
- **Así como en listas `insert(p,x)` *invalida* las posiciones después de `p` (inclusive), en el caso de árboles, una inserción en el nodo `n` invalida las posiciones que son *descendientes* de `n` y que están *a la derecha* de `n`.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

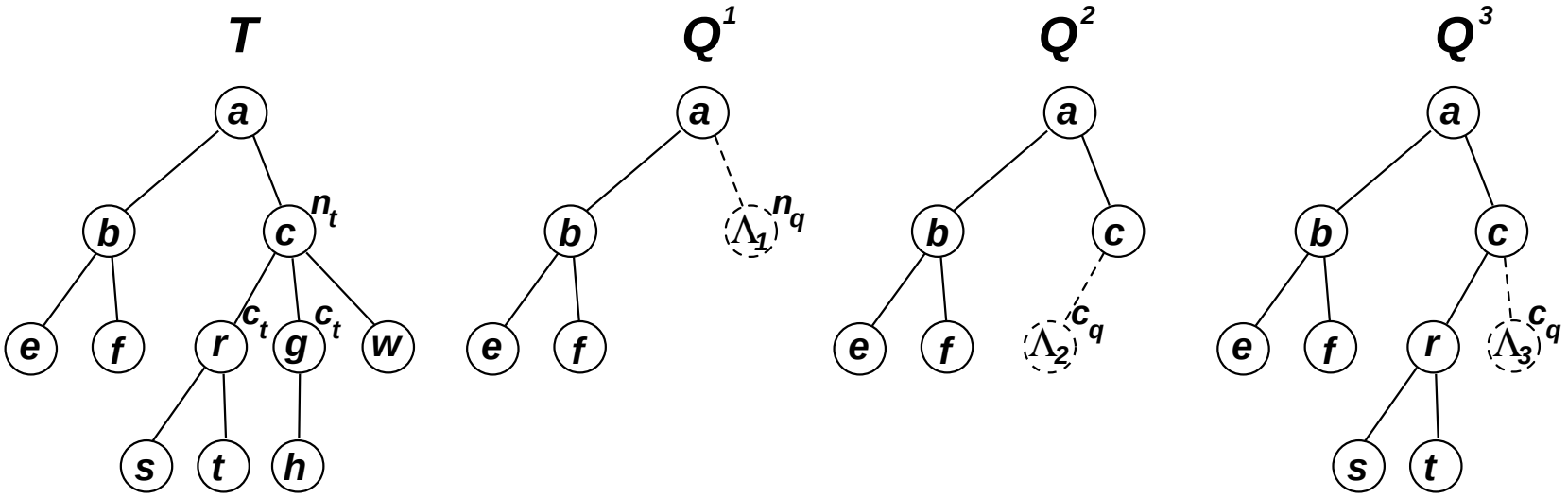
Inserción en árboles (cont.)

Ej: Inserta z en Λ_3 y en g



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmo para copiar árboles



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmo para copiar árboles (cont.)

```
1  iterator tree_copy(tree &T,iterator nt,
2                     tree &Q,iterator nq) {
3    nq = /* nodo resultante de insertar el
4            elemento de 'nt' en 'nq' . . . */;
5    iterator
6      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8    while (/* 'ct' no es 'Lambda'. . . */) {
9      cq = tree_copy(T,ct,Q,cq);
10     ct = /* hermano derecho de 'ct'. . . */;
11     cq = /* hermano derecho de 'cq'. . . */;
12   }
13   return nq;
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmo para copiar árboles espejado

```
1  iterator mirror_copy(tree &T,iterator nt,
2                       tree &Q,iterator nq) {
3    nq = /* nodo resultante de insertar
4            el elemento de 'nt' en 'nq' */;
5    iterator
6      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8    while (/* 'ct' no es 'Lambda'. . . */) {
9      cq = mirror_copy(T,ct,Q,cq);
10     ct = /* hermano derecho de 'ct' . . . */;
11   }
12   return nq;
13 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Supresión en árboles

- **Al igual que en listas, sólo se puede suprimir en posiciones *dereferenciables*.**
- **En el caso de suprimir en un nodo *hoja*, sólo se elimina el nodo.**
- **Si el nodo tiene hijos, eliminarlo equivale a eliminar todo el *subárbol* correspondiente.**
- **Como en listas, eliminando un nodo devuelve la posición del *hermano derecho* que llena el espacio dejado por el nodo eliminado.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Supresión en árboles (cont.)

"Poda" un árbol, eliminando todos los nodos de un árbol que son impares *incluyendo sus subárboles.* Por ejemplo, si `T=(6 (2 3 4) (5 8 10))`. Entonces después de aplicar `prune_odd` tenemos `T=(6 (2 4))`. Notar que los nodos 8 y 10 se eliminan porque son hijos de 5 que es impar.

```
1  iterator_t prune_odd(tree &T,iterator_t n) {
2    if (/*valor de 'n'. . . */ % 2)
3      /* elimina el nodo 'n' y refresca. . . */;
4    else {
5      iterator_t c =
6        /* hijo mas izquierdo de 'n' . . . */;
7      while (/*'c' no es 'Lambda' . . . */)
8        c = prune_odd(T,c);
9      n = /* hermano derecho de 'n'. . . */;
10   }
11   return n;
12 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones básicas sobre árboles

- **Dado un nodo (posición o iterator sobre el árbol), obtener su *hijo más izquierdo*. (Puede retornar una posición Λ).**
- **Dado un nodo obtener su *hermano derecho*. (Puede retornar una posición Λ).**
- **Dada una posición, determinar si es Λ o no.**
- **Obtener la posición de la *raíz* del árbol.**
- **Dado un nodo obtener una referencia al *dato* contenido en el nodo.**
- **Dada una posición (dereferenciable o no) y un dato, insertar un nuevo nodo con ese dato en esa posición.**
- ***Borrar* un nodo y todo su subárbol correspondiente.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación de árboles

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para árboles

```
1  class iterator_t {
2    /* . . . . */
3  public:
4    iterator_t lchild();
5    iterator_t right();
6  };
7
8  class tree {
9    /* . . . . */
10 public:
11   iterator_t begin();
12   iterator_t end();
13   elem_t &retrieve(iterator_t p);
14   iterator_t insert(iterator_t p,elem_t t);
15   iterator_t erase(iterator_t p);
16   void clear();
17   iterator_t splice(iterator_t to,iterator_t from);
18 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para árboles (cont.)

Como con las listas y correspondencias tenemos una clase `iterator_t` que nos permite iterar sobre los nodos del árbol, tanto dereferenciables como no dereferenciables. En lo que sigue `T` es un árbol, `p`, `q` y `r` son nodos (iterators) y `x` es un elemento de tipo `elem_t`.

- **`q = p.lchild()`: Dada una posición dereferenciable `p` retorna la posición del hijo más izquierdo (*"leftmost child"*). La posición retornada puede ser dereferenciable o no.**
- **`q = p.right()`: Dada una posición dereferenciable `p` retorna la posición del hermano derecho. La posición retornada puede ser dereferenciable o no.**
- **`T.end()`: retorna un iterator no dereferenciable.**
- **`p=begin()`: retorna la posición del comienzo del árbol, es decir la raíz. Si el árbol está vacío, entonces retorna `end()`.**

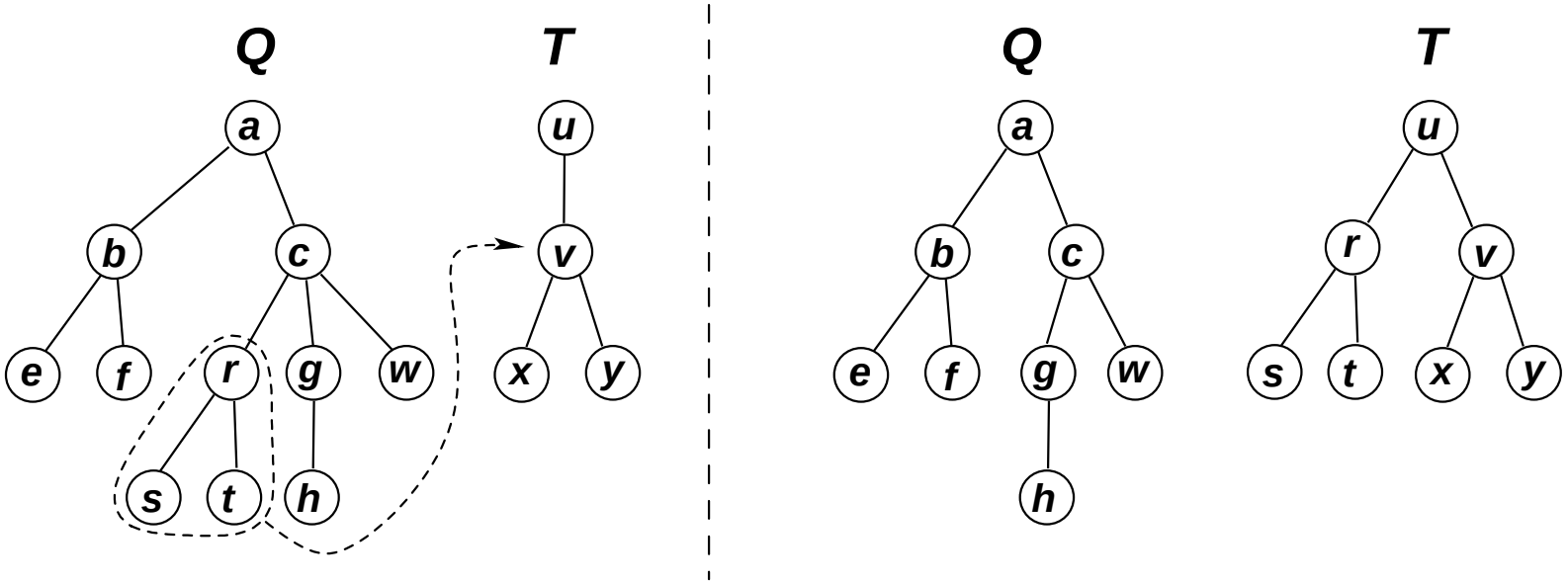
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para árboles (cont.)

- `x = T.retrieve(p)`: Dada una posición dereferenciable retorna una referencia al valor correspondiente.
- `q = T.insert(p,x)`: Inserta un nuevo nodo en la posición `p` conteniendo el elemento `x`. `p` puede ser dereferenciable o no. Retorna la posición del nuevo elemento insertado.
- `p = T.erase(p)`: Elimina la posición dereferenciable `p` y todo el subárbol de `p`.
- `T.clear()`: Elimina todos los elementos del árbol (equivale a `T.erase(T.begin())`).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para árboles (cont.)



- `T.splice(to,from)`: **Elimina todo el subárbol del nodo dereferenciable** `from` **y lo inserta en el nodo (dereferenciable o no)** `to`**.** `to`**,** `from` **pueden estar en diferentes árboles. Si están en el mismo árbol** `to` **no puede ser descendiente de** `from`**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listado orden previo

```
1  void preorder(tree &T,iterator_t n,list<int> &L) {
2    L.insert(L.end(),T.retrieve(n));
3
4    iterator_t c = n.lchild();
5    while (c!=T.end()) {
6      preorder(T,c,L);
7      c = c.right();
8    }
9  }
10 void preorder(tree &T,list<int> &L) {
11   if (T.begin()==T.end()) return;
12   preorder(T,T.begin(),L);
13 }
14
15 //---:---<*>---:---<*>---:---<*>---:---<*>
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listado orden posterior

```
1  void postorder(tree &T,iterator_t n,list<int> &L) {
2    iterator_t c = n.lchild();
3    while (c!=T.end()) {
4      postorder(T,c,L);
5      c = c.right();
6    }
7    L.insert(L.end(),T.retrieve(n));
8  }
9  void postorder(tree &T,list<int> &L) {
10   if (T.begin()==T.end()) return;
11   postorder(T,T.begin(),L);
12 }
13
14 //---:---<*>---:---<*>---:---<*>---:---<*>
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp

```
1  void lisp_print(tree &T,iterator_t n) {
2    iterator_t c = n.lchild();
3    if (c==T.end()) cout << T.retrieve(n);
4    else {
5      cout << "(" << T.retrieve(n);
6      while (c!=T.end()) {
7        cout << " ";
8        lisp_print(T,c);
9        c = c.right();
10     }
11     cout << ")";
12   }
13 }
14 void lisp_print(tree &T) {
15   if (T.begin()!=T.end()) lisp_print(T,T.begin());
16 }
17
18 //---:---<*>---:---<*>---:---<*>---:---<*>
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Copia

```
1  iterator_t tree_copy(tree &T,iterator_t nt,
2                       tree &Q,iterator_t nq) {
3    nq = Q.insert(nq,T.retrieve(nt));
4    iterator_t
5      ct = nt.lchild(),
6      cq = nq.lchild();
7    while (ct!=T.end()) {
8      cq = tree_copy(T,ct,Q,cq);
9      ct = ct.right();
10     cq = cq.right();
11   }
12   return nq;
13 }
14
15 void tree_copy(tree &T,tree &Q) {
16   if (T.begin() != T.end())
17     tree_copy(T,T.begin(),Q,Q.begin());
18 }
19
20 //---:---<*>---:---<*>---:---<*>---:---<*>
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Copia espejo

```
1  iterator_t mirror_copy(tree &T,iterator_t nt,
2                         tree &Q,iterator_t nq) {
3    nq = Q.insert(nq,T.retrieve(nt));
4    iterator_t
5      ct = nt.lchild(),
6      cq = nq.lchild();
7    while (ct != T.end()) {
8      cq = mirror_copy(T,ct,Q,cq);
9      ct = ct.right();
10   }
11   return nq;
12 }
13
14 void mirror_copy(tree &T,tree &Q) {
15   if (T.begin() != T.end())
16     mirror_copy(T,T.begin(),Q,Q.begin());
17 }
18
19 //---:---<*>---:---<*>---:---<*>---:---<*>
```

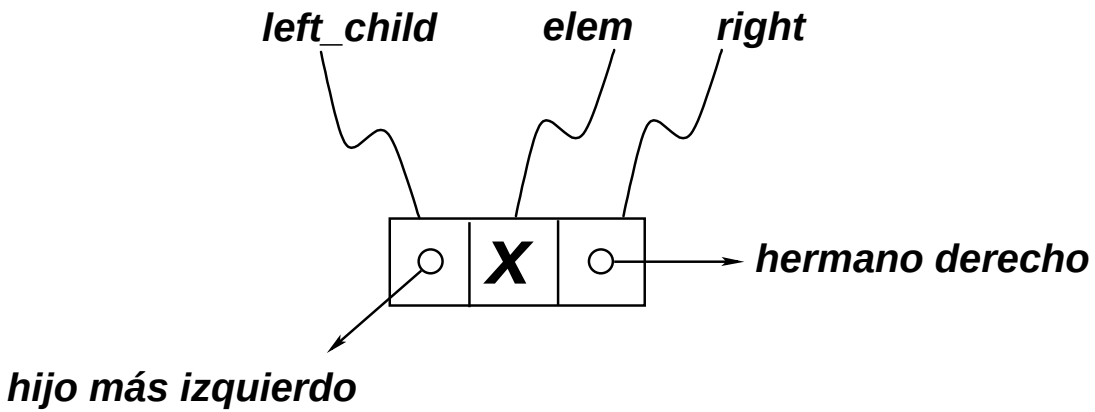

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Poda impares

```
1  iterator_t prune_odd(tree &T,iterator_t n) {
2    if (T.retrieve(n) % 2) n = T.erase(n);
3    else {
4      iterator_t c = n.lchild();
5      while (c != T.end()) c = prune_odd(T,c);
6      n = n.right();
7    }
8    return n;
9  }
10
11 void prune_odd(tree &T) {
12   if (T.begin() != T.end()) prune_odd(T,T.begin());
13 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

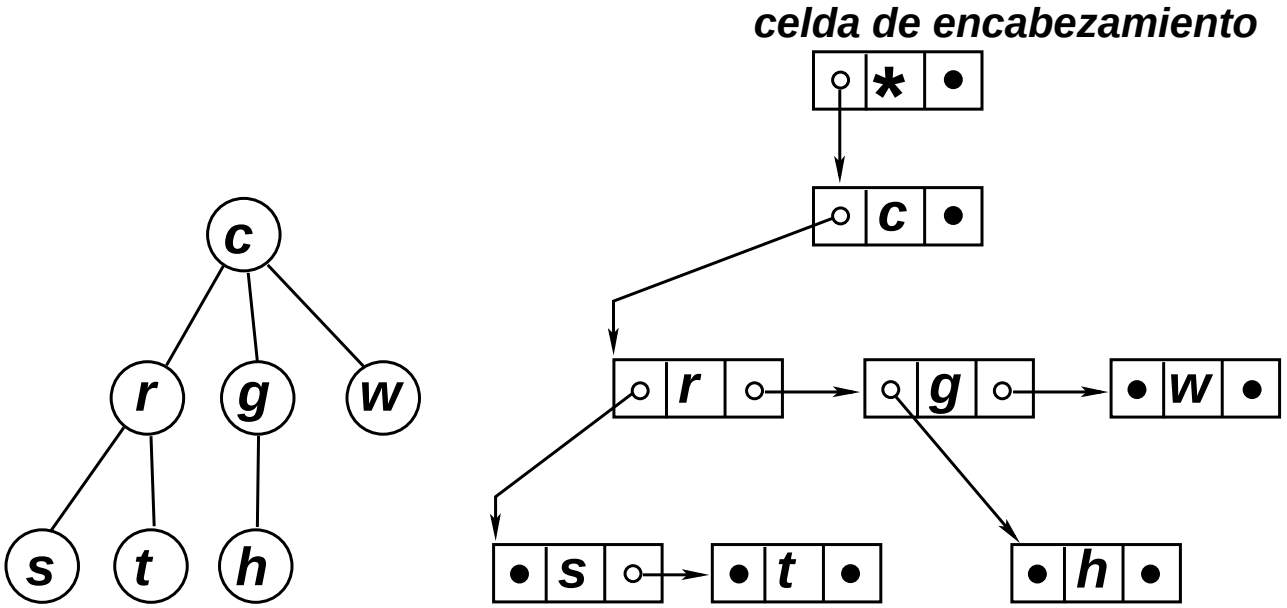
Implementación por punteros



- **Las celdas contienen, además del dato `elem`, un puntero `right` a la celda que corresponde al hermano derecho y otro `left_child` al hijo más izquierdo .**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

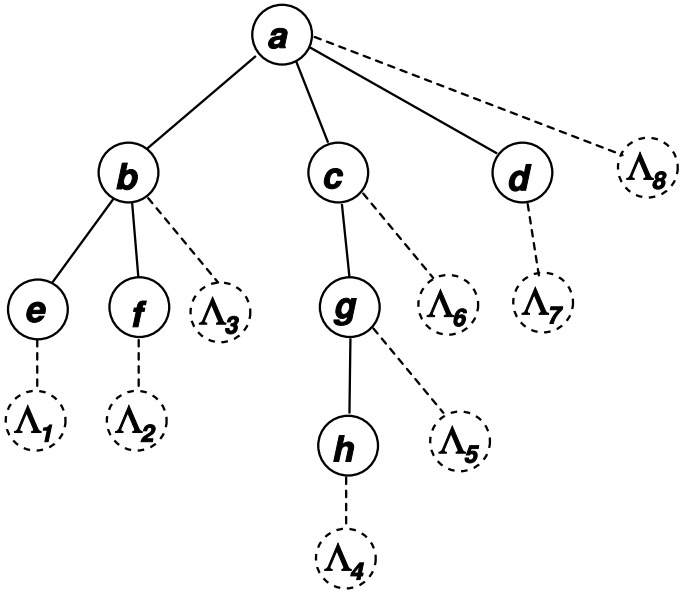
Implementación por punteros (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El tipo iterator

- **Podríamos definir `iterator_t` como un `typedef` a `cell *`.**
- **OK para posiciones dereferenciables y posiciones no dereferenciables Λ_3.**

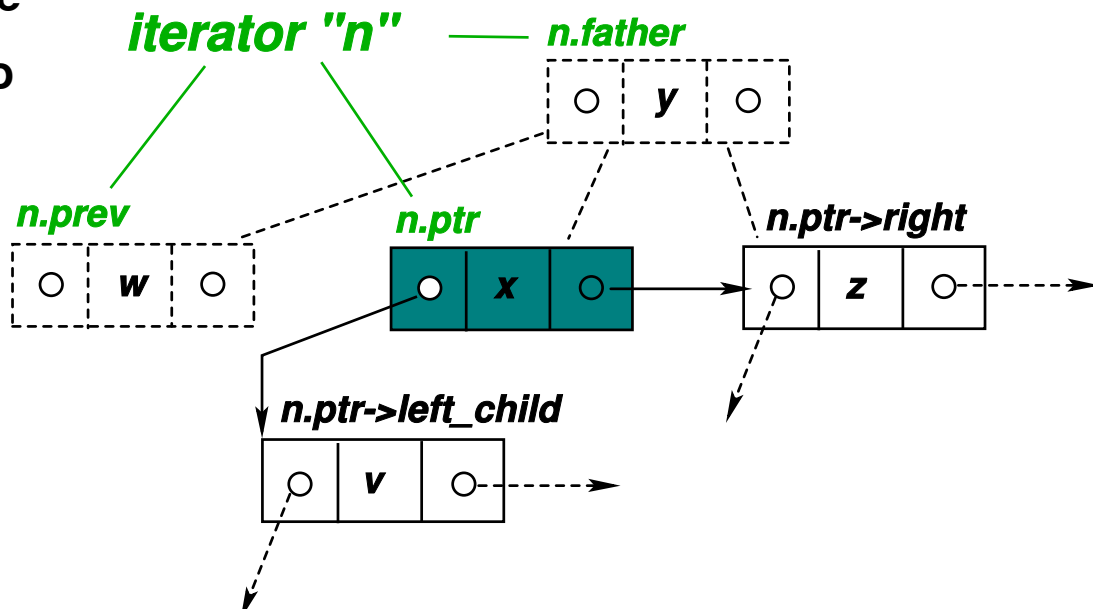


- **Por supuesto habría que mantener el criterio de usar *"posiciones adelantadas"* con respecto al dato.**
- **Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ_1 que provienen de aplicar `lchild()` a una hoja.**

El tipo iterator (cont.)

Iterator consiste en *tres punteros a celdas*

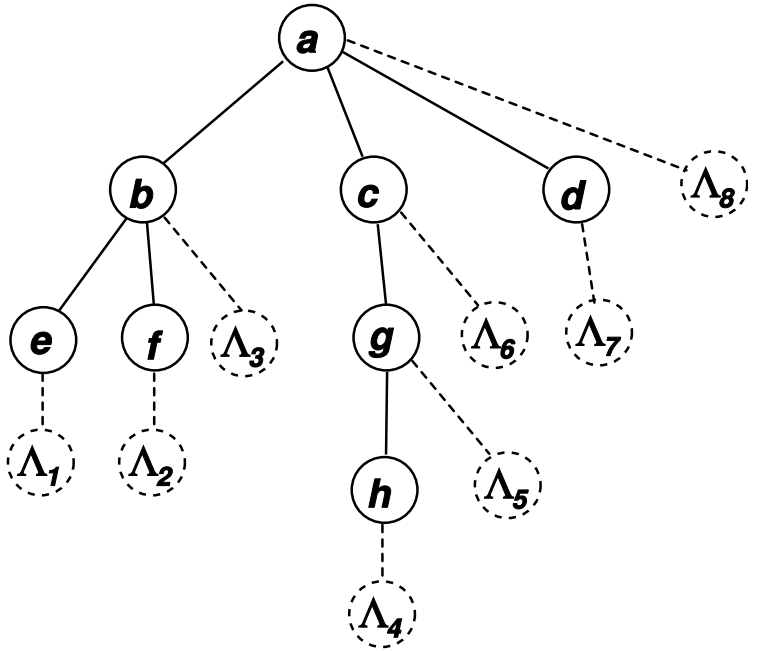
- **Un puntero `ptr` a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables).**
- **Un puntero `prev` al *hermano izquierdo*.**
- **Un puntero `father` al padre.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El tipo iterator (cont.)

- **nodo** e: ***ptr=***e, ***prev=NULL***, ***father=***b.
- **nodo** f: ***ptr=***f, ***prev=***e, ***father=***b.
- **nodo** Λ_1: ***ptr=NULL***, ***prev=NULL***, ***father=***e.
- **nodo** Λ_2: ***ptr=NULL***, ***prev=NULL***, ***father=***f.
- **nodo** Λ_3: ***ptr=NULL***, ***prev=***f, ***father=***b.
- **nodo** g: ***ptr=***g, ***prev=NULL***, ***father=***c.
- **nodo** Λ_6: ***ptr=NULL***, ***prev=***g, ***father=***c.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros

```
1   class tree;
2   class iterator_t;
3
4   //---:---<*>---:---<*>---:---<*>---:---<*>
5   class cell {
6     friend class tree;
7     friend class iterator_t;
8     elem_t elem;
9     cell *right, *left_child;
10    cell() : right(NULL), left_child(NULL) {}
11  };
12
13  //---:---<*>---:---<*>---:---<*>---:---<*>
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1  class iterator_t {
2  private:
3    friend class tree;
4    cell *ptr,*prev,*father;
5    iterator_t(cell *p,cell *prev_a, cell *f_a)
6      : ptr(p), prev(prev_a), father(f_a) { }
7  public:
8    iterator_t(const iterator_t &q) {
9      ptr = q.ptr;
10     prev = q.prev;
11     father = q.father;
12   }
```

- **Constructor privado** `iterator_t(cell *p,cell *pv, cell *f)`
- **Constructor público** `iterator_t(const iterator_t &q)`**. Llamado** *"constructor por copia"* **es utilizado cuando hacemos asignaciones de nodos, por ejemplo** `iterator_t p; p=...; iterator_t q(p);`**.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1    bool operator!=(iterator_t q) { return ptr!=q.ptr; }
2    bool operator==(iterator_t q) { return ptr==q.ptr; }
3    iterator_t()
4      : ptr(NULL), prev(NULL), father(NULL) { }
```

- Los operadores `!=` y `==` nos permitirá comparar nodos por igualdad o desigualdad (`p==q` o `p!=q`).
- Los nodos *no* pueden compararse con los operadores de relación de orden (`<`, `<=`, `>` y `>=`).
- Sólo comparan `ptr` de forma que *todas las posiciones no dereferenciables (Λ) son "iguales" entre sí*. Esto permite comparar en los lazos cualquier posición `p` con `end()`, entonces `p==end()` retornará `true` incluso si `p` no es exactamente igual a `end()` (es decir tiene campos `father` y `prev` diferentes).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

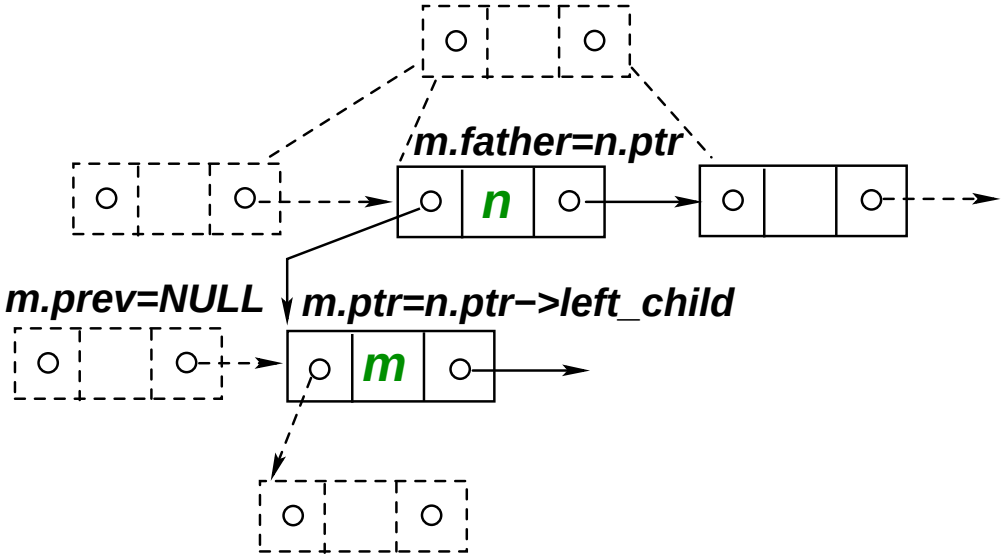
Implementación por punteros (cont.)

```
1     iterator_t lchild() {
2       return iterator_t(ptr->left_child,NULL,ptr);
3     }
4     iterator_t right() {
5       return iterator_t(ptr->right,ptr,father);
6     }
7   };
8
9   //---:---<*>---:---<*>---:---<*>---:---<*>
```

- `lchild()` y `right()` permiten *"movernos"* dentro del árbol.
- Sólo pueden aplicarse a posiciones dereferenciables y pueden retornar posiciones dereferenciables o no.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

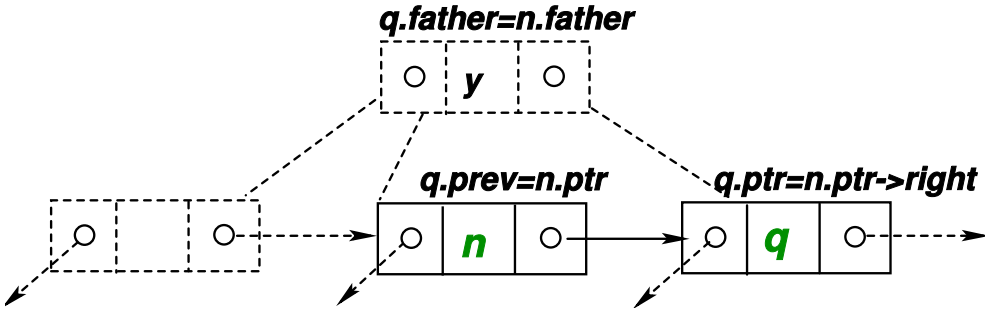
Implementación por punteros (cont.)



- `m = n.lchild()`
- `m.ptr = n.ptr->left_child`
- `m.prev = NULL` (ya que `m` es un *hijo más izquierdo*)
- `m.father = n.ptr`

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)



- `q = n.right()`
- `q.ptr = n.ptr->right`
- `q.prev = n.ptr`
- `q.father = n.father`

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1   class tree {
2   private:
3     cell *header;
4     tree(const tree &T) {}
5   public:
6
7     tree() {
8       header = new cell;
9       header->right = NULL;
10      header->left_child = NULL;
11    }
12    ~tree() { clear(); delete header; }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1   elem_t &retrieve(iterator_t p) {
2     return p.ptr->elem;
3   }
4
5   iterator_t insert(iterator_t p,elem_t elem) {
6     assert(!(p.father==header && p.ptr));
7     cell *c = new cell;
8     c->right = p.ptr;
9     c->elem = elem;
10    p.ptr = c;
11    if (p.prev) p.prev->right = c;
12    else p.father->left_child = c;
13    return p;
14  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1   iterator_t erase(iterator_t p) {
2     if(p==end()) return p;
3     iterator_t c = p.lchild();
4     while (c!=end()) c = erase(c);
5     cell *q = p.ptr;
6     p.ptr = p.ptr->right;
7     if (p.prev) p.prev->right = p.ptr;
8     else p.father->left_child = p.ptr;
9     delete q;
10    return p;
11  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1   iterator_t splice(iterator_t to,iterator_t from) {
2     assert(!(to.father==header && to.ptr));
3     cell *c = from.ptr;
4
5     if (from.prev) from.prev->right = c->right;
6     else from.father->left_child = c->right;
7
8     c->right = to.ptr;
9     to.ptr = c;
10    if (to.prev) to.prev->right = c;
11    else to.father->left_child = c;
12
13    return to;
14  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1   iterator_t find(elem_t elem) {
2     return find(elem,begin());
3   }
4   iterator_t find(elem_t elem,iterator_t p) {
5     if(p==end() || retrieve(p) == elem) return p;
6     iterator_t q,c = p.lchild();
7     while (c!=end()) {
8       q = find(elem,c);
9       if (q!=end()) return q;
10      else c = c.right();
11    }
12    return iterator_t();
13  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por punteros (cont.)

```
1    void clear() { erase(begin()); }
2    iterator_t begin() {
3      return iterator_t(header->left_child,NULL,header);
4    }
5    iterator_t end() { return iterator_t(); }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada

- La clase `tree` pasa a ser ahora un template, de manera que podremos declarar `tree<int>`, `tree<double>`.
- Las clases `cell` e `iterator` son ahora clases anidadas dentro de `tree`. Externamente se verán como `tree<int>::cell` y `tree<int>::iterator`. Sin embargo, sólo `iterator` es pública y es usada *fuera* de `tree`.
- La dereferenciación de posiciones (nodos) `x=retrieve(p)` se reemplaza por `x=*p`. Para eso debemos *"sobrecargar"* el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos, entonces vamos a querer extraer campos de un elemento almacenado en un nodo, por lo cual debemos hacer `(*p).campo`. Para poder hacer esto usando el operador `->` (es decir `p->campo`) debemos sobrecargar el operador `->`. Ambos operadores devuelven referencias de manera que es posible usarlos en el miembro izquierdo, como en `*p=x` o `p->campo=z`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada (cont.)

- Igual que con la interfase básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=`. También tiene definido el constructor por copia.
- El avance por hermano derecho `p = p.right();` ahora se puede hacer con `p++`, de todas formas mantenemos la función `right()` que a veces resulta ser más compacta. Por ejemplo `q = p.right()` se traduce en `q=p; q++;` en la versión con operadores.
- La función estática `cell_count()`, permite obtener el número total de celdas alocadas por todos las instancias de la clase, e incluye las celdas de encabezamiento. Esta función fue introducida para debugging, normalmente no debería ser usada por los usuarios de la clase. Como es estática puede invocarse como `tree<int>::cell_count()` o también sobre una instancia, `T.cell_count()`.

Interfase avanzada (cont.)

- Se ha incluido el operador de asignación, y un constructor por copia, de manera que se puede copiar árboles usando directamente el operador `=`, por ejemplo

```
1   tree<int> T,Q1;
2   // carga elementos en T . . . .
3   Q1 = T;
4   tree<int> Q2(T);
```

Así como también pasar árboles por copia y definir contenedores que contienen árboles como por ejemplo una lista de árboles de enteros. `list< tree<int> >`. Esta función necesita otra función recursiva auxiliar que hemos llamado `tree_copy_aux()` y que normalmente no debería ser usada directamente por los usuarios de la clase, de manera que la incluimos en la sección privada.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso de la interfase avanzada

```
1  typedef tree<int> tree_t;
2  typedef tree_t::iterator node_t;
3
4  int count_nodes(tree_t &T,node_t n) {
5    if (n==T.end()) return 0;
6    int m=1;
7    node_t c = n.lchild();
8    while(c!=T.end()) m += count_nodes(T,c++);
9    return m;
10 }
11
12 int count_nodes(tree_t &T) {
13   return count_nodes(T,T.begin());
14 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso de la interfase avanzada (cont.)

```
1  int leaf_count(tree_t &T,node_t n) {
2    if (n==T.end()) return 0;
3    node_t c = n.lchild();
4    if (c==T.end()) return 1;
5    int w = 0;
6    while (c!=T.end()) w += leaf_count(T,c++);
7    return w;
8  }
9
10 int leaf_count(tree_t &T) {
11   return leaf_count(T,T.begin());
12 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución

- Todas las funciones básicas tienen costo $O(1)$. (incluso `splice()`!!)
- Esto se debe a que la operación de mover todo el árbol de una posición a otra se realiza con una operación de punteros.

Operación	$T(n)$
`begin()`, `end()`, `n.right()`, `n++`, `n.lchild()`, `*n`, `insert()`, `splice(to,from)`	$O(1)$
`erase()`, `find()`, `clear()`, `T1=T2`	$O(n)$

- Las operaciones que no son $O(1)$ son `erase(p)` que debe eliminar todos los nodos del subárbol del nodo `p`, `clear()` que equivale a `erase(begin())`, `find(x)` y el constructor por copia (`T1=T2`). En todos los casos n es o bien el número de nodos del subárbol (`erase(p)` y `find(x,p)`) o bien el número total de nodos del árbol (`clear()`, `find(x)` y el constructor por copia `T1=T2`).

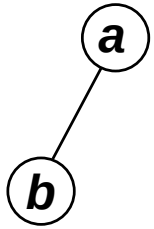
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios

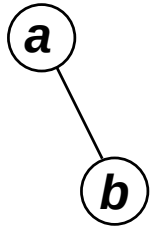
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios

ordenado orientado



binario con hijo izquierdo



binario con hijo derecho

- **Los árboles estudiados son *"árboles ordenados orientados"* (AOO) ya que los hermanos están ordenados entre sí y hay una orientación de los caminos desde la raíz a las hojas.**
- **En el *"árbol binario"* (AB) cada nodo puede tener a lo sumo dos hijos. Además, si un dado nodo n tiene un sólo hijo, entonces este puede ser el hijo derecho o el hijo izquierdo de n.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Listado en orden simétrico

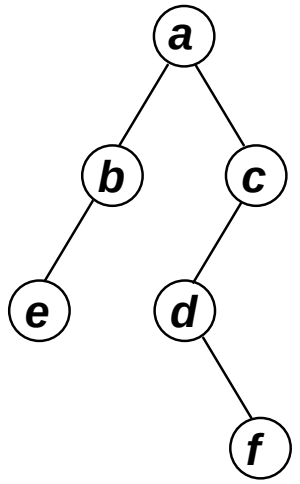
Los listados en orden previo y posterior para AB coinciden con su versión correspondiente para AOO. El *"listado en orden simétrico"* se define recursivamente como

$$\mathrm{osim}(\Lambda) = < \textbf{lista vacía} >$$

$$\mathrm{osim}(n) = \mathrm{osim}(s_l), n, \mathrm{osim}(s_r)$$

donde $n_{l,r}$ son los hijos izquierdo y derecho de n, respectivamente.

Listado en orden simétrico: e, b, a, d, f, c



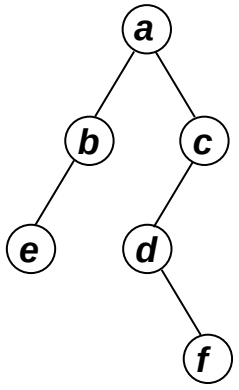
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación Lisp

En el caso en que un nodo tiene un sólo hijo, reemplazamos con un punto la posición del hijo faltante. s_l, s_r hijos izquierdo/derecho.

$$\text{lisp}(n) = \begin{cases} n & ; \textbf{ si } s_l = \Lambda \textbf{ y } s_r = \Lambda \\ (n\ \text{lisp}(s_l)\ \text{lisp}(s_r)) & ; \textbf{ si } s_l \neq \Lambda \textbf{ y } s_r \neq \Lambda \\ (n\ \cdot\ \text{lisp}(s_r)) & ; \textbf{ si } s_r \neq \Lambda \\ (n\ \text{lisp}(s_l)\ \cdot) & ; \textbf{ si } s_l \neq \Lambda \end{cases}$$

$$\text{lisp}(a) = (a\ (b\ e\ \cdot)\ (c\ (d\ \cdot\ f)\ \cdot))$$



Operaciones abstractas sobre árboles

- **Difieren de las de AOO en las funciones que permiten *"moverse"* en el árbol. Existen dos operaciones independientes *"hijo-izquierdo"* e *"hijo-derecho"*.**
- **También, en AB *sólo se puede insertar en un nodo* Λ**

Entonces:

- **Dado un nodo *dereferenciable*, obtener su *hijo izquierdo*. (Puede retornar una posición Λ).**
- **Idem para el *hijo derecho*.**
- **Dada una posición, determinar si es Λ o no.**
- **Obtener la posición de la raíz del árbol.**
- **Dado un nodo obtener una referencia al dato contenido en el nodo.**
- **Dada una posición no dereferenciable y un dato, insertar un nuevo nodo con ese dato en esa posición.**
- **Borrar un nodo y todo su subárbol correspondiente.**

Notar que sólo cambian las dos primeras con respecto a las de AOO.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica

```
1  class iterator_t {
2    /* . . . */
3   public:
4    iterator_t left();
5    iterator_t right();
6  };
7
8  class btree {
9    /* . . . */
10  public:
11   iterator_t begin();
12   iterator_t end();
13   elem_t & retrieve(iterator_t p);
14   iterator_t insert(iterator_t p,elem_t t);
15   iterator_t erase(iterator_t p);
16   void clear();
17   iterator_t splice(iterator_t to,iterator_t from);
18 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso. Predicado de igualdad

```
1  bool equal_p (btree &T,iterator_t nt,
2                btree &Q,iterator_t nq) {
3    if (nt==T.end() xor nq==Q.end()) return false;
4    if (nt==T.end()) return true;
5    if (T.retrieve(nt) != Q.retrieve(nq)) return false;
6    return equal_p(T,nt.right(),Q,nq.right()) &&
7      equal_p(T,nt.left(),Q,nq.left());
8  }
9  bool equal_p(btree &T,btree &Q) {
10   return equal_p(T,T.begin(),Q,Q.begin());
11 }
```

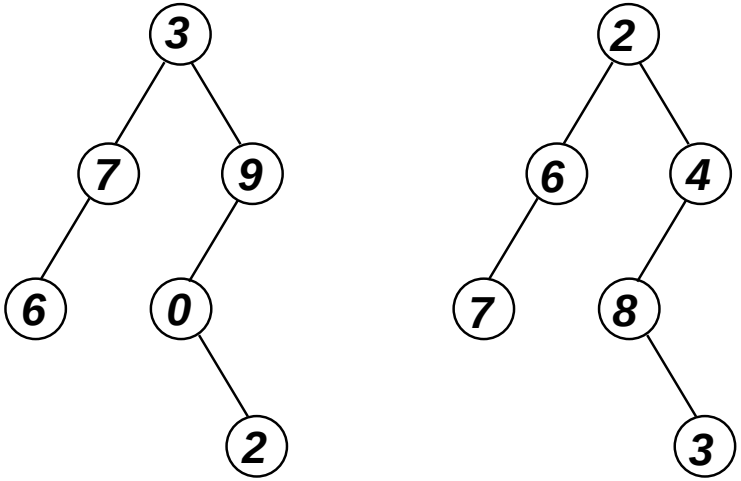
Función predicado (retorna un valor booleano) que determina si dos árboles binarios `T` y `Q` son iguales. Recursivamente dos árboles son iguales si

- **Ambos son vacíos**
- **Ambos no son vacíos, los valores de sus nodos son iguales y los hijos respectivos de su nodo raíz son iguales.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Semejante

Modificando ligeramente se verifica si son iguales en cuanto a su estructura, sin tener en cuenta el valor de los nodos.



- **Ambos son vacíos**
- **Ambos no son vacíos, y los hijos respectivos de su nodo raíz son semejantes.**

Notar que la única diferencia es que no se comparan los valores de las raíces de los subárboles comparados.

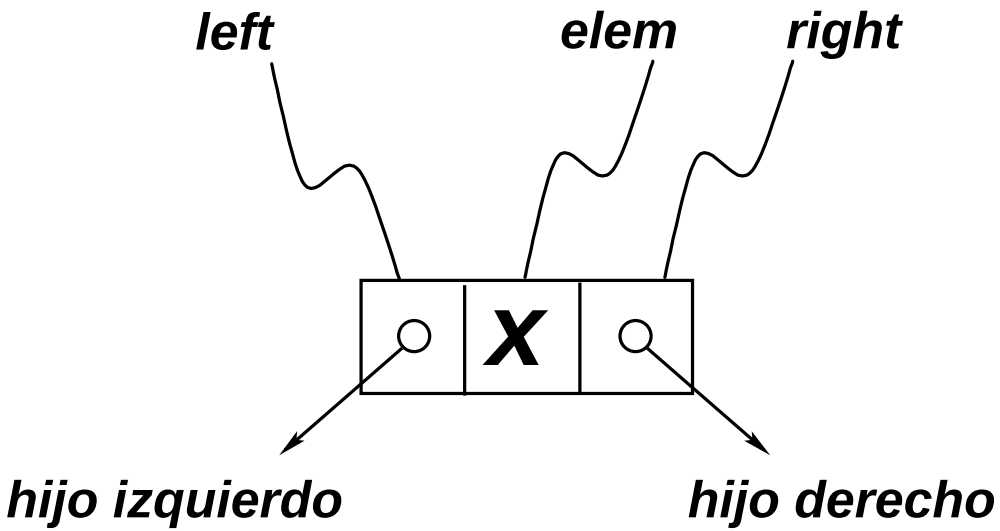
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Semejante (cont.)

```
1  bool semejante_p (btree &T,iterator_t nt,
2                    btree &Q,iterator_t nq) {
3    if (nt==T.end() xor nq==Q.end()) return false;
4    if (nt==T.end()) return true;
5    return semejante_p(T,nt.right(),Q,nq.right()) &&
6      semejante_p(T,nt.left(),Q,nq.left());
7  }
8  bool semejante_p(btree &T,btree &Q) {
9    return semejante_p(T,T.begin(),Q,Q.begin());
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

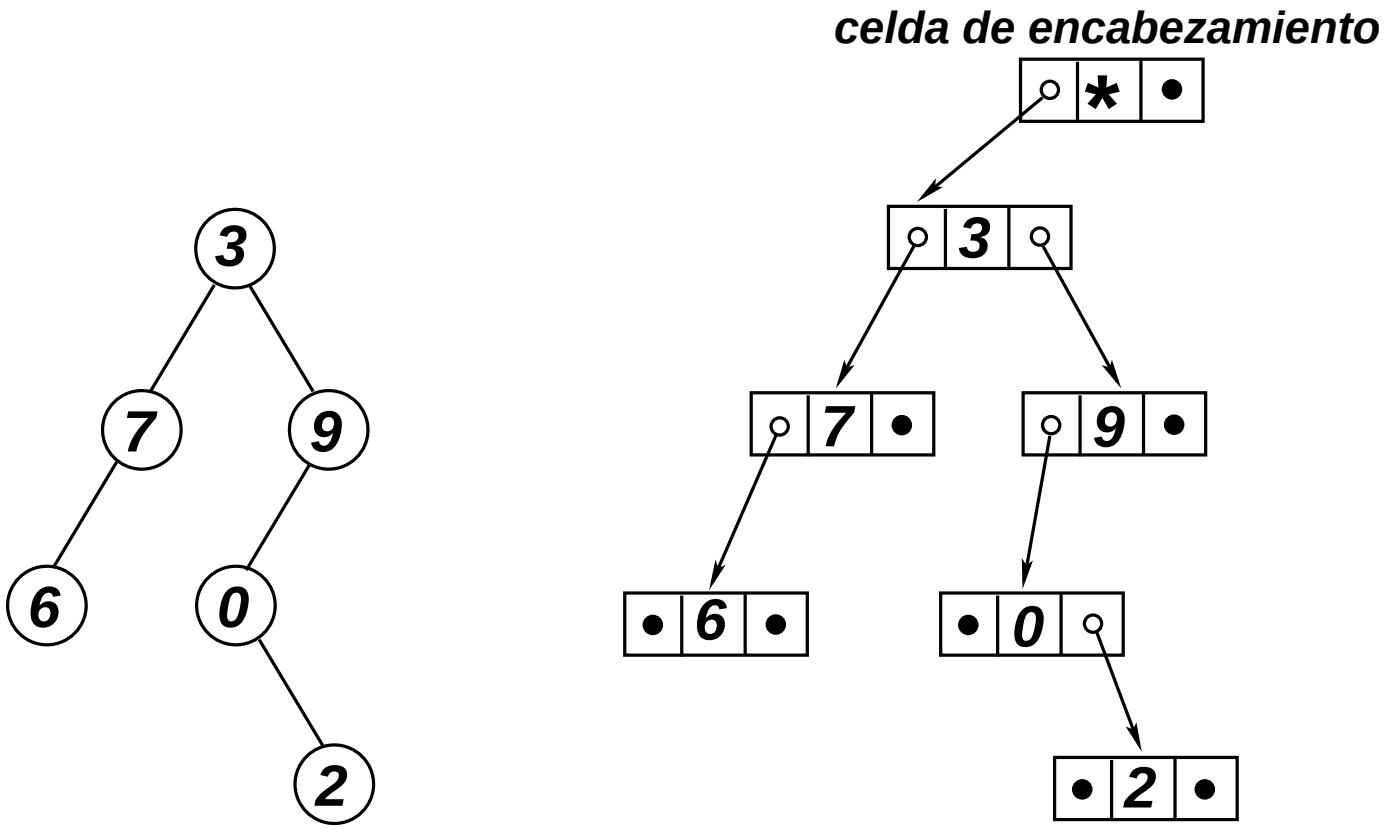
Implementación por punteros



Así como la diferencia entre las interfases de AOO y AB difieren en las funciones `lchild()` y `right()` que son reemplazadas por `left()` y `right()`. (Recordar que `right()` tiene significado diferente en AOO y AB.) Esto induce naturalmente a considerar que en la celda haya dos punteros que apunten al hijo izquierdo y al hijo derecho.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

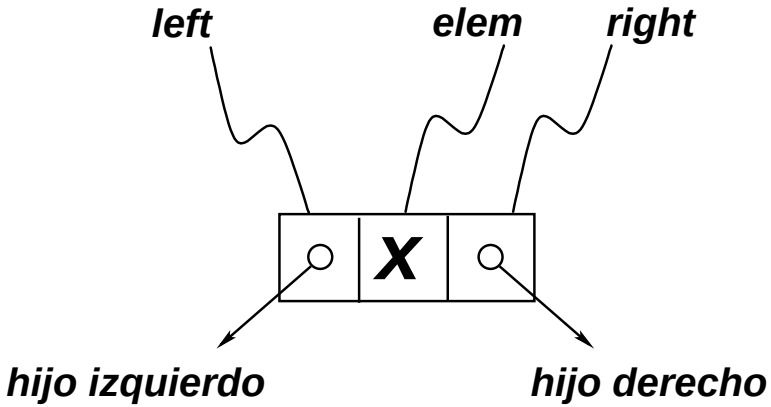
Implementación por punteros (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase cell

```
1  class cell {
2    friend class btree;
3    friend class iterator_t;
4    elem_t t;
5    cell *right,*left;
6    cell() : right(NULL), left(NULL) {}
7  };
```



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase iterator

```
1  class iterator_t {
2  private:
3    friend class btree;
4    cell *ptr,*father;
5    enum side_t {NONE,R,L};
6    side_t side;
7    iterator_t(cell *p,side_t side_a,cell *f_a)
8      : ptr(p), side(side_a), father(f_a) { }
```

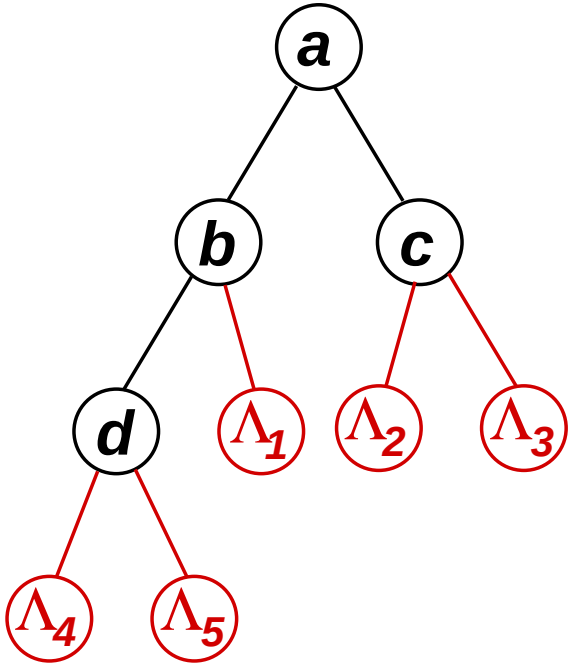
- Contiene un puntero a la celda, y otro al padre, como en el caso del AOO.
- El puntero `prev` que apunta al hermano a la izquierda, aquí ya no tiene sentido. Recordemos que el iterator nos debe permitir ubicar a las posiciones, incluso aquellas que son Λ. Para ello incluimos el iterator un miembro `side` de tipo `enum side_t`, que puede tomar los valores `R` (right) y `L` (left).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase iterator (cont.)

- **nodo** b**:** ***ptr*=**b**,** ***father*=**a**,** ***side=L***
- **nodo** c**:** ***ptr*=**c**,** ***father*=**a**,** ***side=R***
- **nodo** d**:** ***ptr*=**d**,** ***father*=**b**,** ***side=L***
- **nodo** Λ_1**:** ***ptr=NULL*, *father*=**b**,** ***side=R***
- **nodo** Λ_2**:** ***ptr=NULL*, *father*=**c**,** ***side=L***
- **nodo** Λ_3**:** ***ptr=NULL*, *father*=**c**,** ***side=R***
- **nodo** Λ_4**:** ***ptr=NULL*, *father*=**d**,** ***side=L***
- **nodo** Λ_5**:** ***ptr=NULL*, *father*=**d**,** ***side=R***

Notar que si not tuviéramos el campo ***side*****,** Λ_4 **y** Λ_5 **serían iguales.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase iterator (cont.)

```
1  public:
2    iterator_t(const iterator_t &q) {
3      ptr = q.ptr;
4      side = q.side;
5      father = q.father;
6    }
7    bool operator!=(iterator_t q) { return ptr!=q.ptr; }
8    bool operator==(iterator_t q) { return ptr==q.ptr; }
```

- La comparación de iterators compara sólo los campos `ptr`, de manera que todos los iterators Λ resultan iguales entre sí (ya que tienen `ptr=NULL`). Como `end()` retorna un iterator Λ (ver más abajo), entonces esto habilita a usar las comparaciones típicas

```
1  if (c!=T.end()) {
2    // . . . .
3  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase iterator (cont.)

```
1     iterator_t() : ptr(NULL), side(NONE),
2
3                    father(NULL) { }
```

- El constructor por defecto de la clase `iterator` retorna un iterator no dereferenciable que no existe en el árbol. Todos sus punteros son nulos y `side` es un valor especial de `side_t` llamado `NONE`. Insertar en este iterator es un error.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree

```
1     iterator_t left() {
2       return iterator_t(ptr->left,L,ptr);
3     }
4     iterator_t right() {
5       return iterator_t(ptr->right,R,ptr);
6     }
7   };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1   class btree {
2   private:
3     cell *header;
4     iterator_t tree_copy_aux(iterator_t nq,
5                              btree &TT,iterator_t nt) {
6       nq = insert(nq,TT.retrieve(nt));
7       iterator_t m = nt.left();
8       if (m != TT.end()) tree_copy_aux(nq.left(),TT,m);
9       m = nt.right();
10      if (m != TT.end()) tree_copy_aux(nq.right(),TT,m);
11      return nq;
12    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1  public:
2    static int cell_count_m;
3    static int cell_count() { return cell_count_m; }
4    btree() {
5      header = new cell;
6      cell_count_m++;
7      header->right = NULL;
8      header->left = NULL;
9    }
```

- La clase `btree` incluye un contador de celdas `cell_count()` y constructor por copia `btree(const btree &)`, como para AOO.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1   btree(const btree &TT) {
2     if (&TT != this) {
3       header = new cell;
4       cell_count_m++;
5       header->right = NULL;
6       header->left = NULL;
7       btree &TTT = (btree &) TT;
8       if (TTT.begin()!=TTT.end())
9         tree_copy_aux(begin(),TTT,TTT.begin());
10    }
11  }
12  ~btree() { clear(); delete header; cell_count_m--; }
13  elem_t & retrieve(iterator_t p) { return p.ptr->t; }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1    iterator_t insert(iterator_t p,elem_t t) {
2      cell *c = new cell;
3      cell_count_m++;
4      c->t = t;
5      if (p.side == iterator_t::R)
6        p.father->right = c;
7      else p.father->left = c;
8      p.ptr = c;
9      return p;
10   }
```

- La diferencia principal está en `insert(p,x)` y `erase(p)`. En insert se crea la celda (actualizando el contador de celdas) y se inserta el dato. Recordar que los campos punteros de la celda quedan en `NULL`, porque así se inicializan en el constructor de celdas. El único campo de celdas que se debe actualizar es, o bien el campo `left` o `right` de la celda padre. Cuál de ellos es el que debe apuntar a la nueva celda se deduce de `p.side` en el iterator. Finalmente se debe actualizar el iterator de forma que `ptr` apunte a la celda creada.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1    iterator_t erase(iterator_t p) {
2      if(p==end()) return p;
3      erase(p.right());
4      erase(p.left());
5      if (p.side == iterator_t::R)
6        p.father->right = NULL;
7      else p.father->left = NULL;
8      delete p.ptr;
9      cell_count_m--;
10     p.ptr = NULL;
11     return p;
12   }
```

- `erase(p)` elimina primero recursivamente todo el subárbol de los hijos izquierdo y derecho de `p`. Después libera la celda actualizando el campo correspondiente del padre (dependiendo de `p.side`). También se actualiza el contador `cell_count_m` al liberar la celda. Notar la actualización del contador por la liberación de las celdas en los subárboles de los hijos se hace automáticamente dentro de la llamada recursiva, de manera que en `erase(p)` sólo hay que liberar explícitamente a la celda `p.ptr`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1   iterator_t splice(iterator_t to,iterator_t from) {
2     cell *c = from.ptr;
3     from.ptr = NULL;
4     if (from.side == iterator_t::R)
5       from.father->right = NULL;
6     else
7       from.father->left = NULL;
8     if (to.side == iterator_t::R) to.father->right = c;
9     else to.father->left = c;
10    to.ptr = c;
11    return to;
12  }
```

- El código de `splice(to,from)` es prácticamente un erase de `from` seguido de un `insert` en `to`.
- `cell_count_m` no cambia, ya que no se crean ni destruyen celdas.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

La clase btree (cont.)

```
1    void clear() { erase(begin()); }
2    iterator_t begin() {
3      return iterator_t(header->left,
4                        iterator_t::L,header);
5    }
6    iterator_t end() { return iterator_t(); }
```

- La posición raíz del árbol se elige como el hijo izquierdo de la celda de encabezamiento. Esto es una convención, podríamos haber elegido también el hijo derecho.
- `end()` retorna un iterator no dereferenciable dado por el constructor por defecto de la clase `iterator_t` (descripto previamente). Este iterator debería ser usado sólo para comparar. Insertar en este iterator es un error.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada

- La clase es un template sobre el tipo contenido en el dato (`class T`) de manera que podremos declarar `btree<int>`, `btree<double>` ...
- La dereferenciación de nodo se hace sobrecargando los operadores `*` y `->`, de manera que podemos hacer

```
1  x = *n;
2  *n = w;
3  y = n->member;
4  n->member = v;
```

  donde `n` es de tipo iterator, `x, w` con de tipo `T` y `member` es algún campo de la clase `T` (si es una clase compuesta). También es válido hacer `n->f(...)` si `f` es un método de la clase `T`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Programación funcional en árboles binarios

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso. Apply (progr. funcional)

- **Podemos escribir algoritmos que modifican los valores de un árbol, por ejemplo sumarle a todos los valores contenidos en un árbol un valor, o duplicarlos.**
- **Todos estos son casos particulares de un algoritmo más general `apply(Q,f)` que tiene como argumentos un árbol `Q` y una *"función escalar"* `T f(T)`. y le aplica a cada uno de los valores nodales la función en cuestión. Este es un ejemplo de *"programación funcional"*, es decir, programación en los cuales los datos de los algoritmos pueden ser también funciones.**
- ***C++* tiene un soporte básico para la programación funcional pasando *"punteros a funciones"*. También se puede usar *"functors"*, clases que sobrecargan el operador `()`.**
- **Lenguajes funcionales puros: Lisp/Scheme, ML, Haskell... las funciones son *"objetos de primera clase"*.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso. Apply (progr. funcional) (cont.)

```
1   template<class T>
2   void apply(btree<T> &Q,
3              typename btree<T>::iterator n,
4              T(*f)(T)) {
5     if (n==Q.end()) return;
6     *n = f(*n);
7     apply(Q,n.left(),f);
8     apply(Q,n.right(),f);
9   }
10  template<class T>
11  void apply(btree<T> &Q,T(*f)(T)) {
12    apply(Q,Q.begin(),f);
```

- Se pasa *el puntero a la función*.
- La *"signatura"* es `T f(T)`. La declaración de punteros a tales funciones se hace reemplazando en la signatura el nombre de la función por `(*f)`.
- Usa una función auxiliar recursiva que toma un argumento adicional de tipo iterator.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ejemplo de uso. Apply (progr. funcional) (cont.)

- **Dentro de esta función auxiliar el puntero a función `f` se aplica como una función normal.**
- **Si `n` es Λ la función simplemente retorna. Si no lo está, entonces aplica la función al valor almacenado en `n` y después llama `apply` recursivamente a sus hijos izquierdo y derecho.**
- **Ejemplo de uso:**

```
1  btree<int> T;
2  int suma10(int j) { return j+10; }
3  apply(T,suma10);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Otros ejemplos de progr. funcional

- `reduce(Q,g)` que toma como argumentos un árbol `Q` y una función asociativa `T g(T,T)` (por ejemplo la suma, el producto, el máximo o el mínimo) y devuelve el resultado de aplicar la función asociativa a todos los valores nodales, hasta llegar a un único valor. Por ejemplo, si hacemos que `g(x,y)` retorne `x+y` retornará la suma de todas las etiquetas del árbol y si hacemos que retorne el máximo, entonces retornará el máximo de todas las etiquetas del árbol.

```
1  int reduce(btree<int> T,int (*f)(int j,int k));
2
3  btree<int> T;
4  int suma(int j,int k) { return j+k; }
5  cout << "Suma de los elementos del arbol: "
6     << reduce(T,suma) << endl;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Otros ejemplos de progr. funcional (cont.)

- Otra aplicación pueden ser *"filtros"*, como la función `prune_odd`. Podríamos escribir una función `remove_if(Q,pred)` que tiene como argumentos un árbol `Q` y una función predicado `bool pred(T)`. La función `remove_if` elimina todos los nodos `n` (y sus subárboles) para cuyos valores la función `pred(*n)` retorna verdadero. La función `prune_odd` se podría obtener entonces simplemente pasando a `remove_if` una función predicado que retorna verdadero si el argumento es impar.

```
1  void remove_if(btree<int> T,bool (*pred)(int j));
2
3  btree<int> T;
4  bool es_impar(int j) { return j%2; }
5  remove_if(T,es_impar);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles de Huffman

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles de Huffman. Intro. a compresión

- **Los árboles de Huffman son un ejemplo interesante de utilización del TAD AB. El objetivo es comprimir archivos o mensajes de texto. Por simplicidad, supongamos que tenemos una cadena de N caracteres compuesta de un cierto conjunto reducido de caracteres C. Por ejemplo si consideramos las letras $C = \{a, b, c, d\}$ entonces el mensaje podría ser $abdcdacabbcdba$. El objetivo es encontrar un representación del mensaje en bits (es decir una cadena de 0's y 1's) lo más corta posible. A esta cadena de 0's y 1's la llamaremos *"el mensaje encodado"*. El algoritmo debe permitir recuperar el mensaje original, ya que de esta forma, si la cadena de caracteres representa un archivo, entonces podemos guardar el mensaje encodado (que es más corto) con el consecuente ahorro de espacio.**

Arboles de Huffman. Intro. a compresión (cont.)

Una primera posibilidad es el código provisto por la representación binaria ASCII de los caracteres. De esta forma cada caracter se encoda en un código de 8 bits. Si el mensaje tiene N caracteres, entonces el mensaje encodado tendrá una longitud de $l = 8N$ bits, resultando en una longitud promedio de

$$\langle l \rangle = \frac{l}{N} = 8 \text{ bits/caracter}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles de Huffman. Intro. a compresión (cont.)

	$C1$	$C2$	$C3$
a	**00**	**0**	**0**
b	**01**	**100**	**01**
c	**10**	**101**	**10**
d	**11**	**11**	**101**

Pero como sabemos que el mensaje sólo está compuesto de las cuatro letras a, b, c, d podemos crear un código de dos bits como el $C1$, de manera que un mensaje como $abcdcba$ se encoda en 00011011100100. Desencodar un mensaje también es simple, vamos tomando dos bits del mensaje y consultando el código lo vamos convirtiendo al carácter correspondiente. En este caso la longitud del código pasa a ser de $\langle l \rangle = 2$ bits/caracter, es decir la cuarta parte del código ASCII.

Arboles de Huffman. Intro. a compresión (cont.)

Por supuesto esta gran compresión se produce por el reducido conjunto de caracteres utilizados. Si el conjunto de caracteres fuera de tamaño 8 en vez de 4 necesitaríamos al menos códigos de 3 bits, resultando en una tasa de 3 bits/caracter. En general si l número de caracteres es n_c la tasa será de

$$\langle l \rangle = \text{ceil}(\log_2 n_c)$$

Si consideramos texto común, el número de caracteres puede oscilar entre unos 90 y 128 caracteres, con lo cual la tasa será de 7 bits por caracter, lo cual representa una ganancia relativamente pequeña del 12.5%.

Arboles de Huffman. Intro. a compresión (cont.)

	$C1$	$C2$	$C3$
a	**00**	**0**	**0**
b	**01**	**100**	**01**
c	**10**	**101**	**10**
d	**11**	**11**	**101**

Utilizar códigos de longitud variable, tratando de asignar códigos de longitud corta a los caracteres que tienen más probabilidad de aparecer y códigos más largos a los que tienen menos probabilidad de aparecer. Por ejemplo, sea $P(a) = 0.7$ **y** $P(b) = P(c) = P(d) = 0.1$**, consideremos** $C2$**. Si bien la longitud en bits de** a **es menor en** $C2$ **que en** $C1$**, la longitud de** b **y** c **es mayor. Un mensaje típico de 100 caracteres contiene 70 a's, 10 b's, 10c's y 10d's. Mensaje encodado tiene longitud:** $70 \times 1 + 10 \times 3 + 10 \times 3 + 10 \times 2 = 150$ **bits, resultando en una longitud promedio de** $\langle l \rangle$**=1.5 bit/caracter.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles de Huffman. Intro. a compresión (cont.)

En general

$$\begin{aligned}\langle l \rangle &= \frac{150 \text{ bits}}{100 \text{ caracteres}} \\ &= 0.70 \times 1 + 0.1 \times 3 + 0.1 \times 3 + 0.1 \times 2 \\ &= \sum_c P(c) l(c)\end{aligned}$$

Esta longitud media representa una compresión de 25% con respecto al $C1$**. Por supuesto, la ventaja del código** $C2$ **se debe a la gran diferencia en probabilidades entre** a **y los otros caracteres. Si la diferencia fuera menor, digamos** $P(a) = 0.4$, $P(b) = P(c) = P(d) = 0.2$**, entonces la longitud de un mensaje típico de 100 caracteres sería** $40 \times 1 + 20 \times 3 + 20 \times 3 + 20 \times 2 = 200$ **bits, o sea una tasa de 2 bits/caracter, igual a la de** $C1$**.**

Condición de prefijos

	$C1$	$C2$	$C3$
a	**00**	**0**	**0**
b	**01**	**100**	**01**
c	**10**	**101**	**10**
d	**11**	**11**	**101**

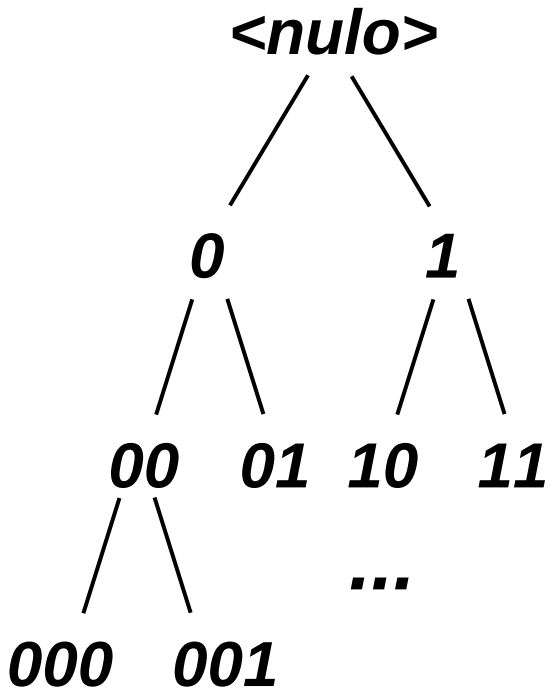
En el caso de que un código de longitud variable como el $C2$ sea más conveniente, debemos poder asegurarnos poder desencodar los mensajes, es decir que la relación entre mensaje y mensaje encodado sea unívoca y que exista un algoritmo para encodar y desencodar mensajes en un tiempo razonable. Por empezar los códigos de los diferentes caracteres deben ser diferentes entre sí, pero esto no es suficiente. Por ejemplo el código $C3$ tiene un código diferente para todos los caracteres, pero los mensaje dba y ccc se encodan ambos como 101010.

Condición de prefijos (cont.)

	$C1$	$C2$	$C3$
a	00	0	0
b	01	100	01
c	10	101	10
d	11	11	101

El error del código $C3$ es que el código de d empieza con el código de c. Decimos que el código de c es *"prefijo"* del de d. Así, al comenzar a desencodar el mensaje 101010. Cuando leemos los dos primeros bits 10, no sabemos si ya extraer una c o seguir con el tercer bit para formar una d. Notar que también el código de a es prefijo del de b. Por lo tanto, una condición para admitir un código es que cumpla con la *"condición de prefijos"*, a saber que *el código de un caracter no sea prefijo del código de ningún otro caracter*. Los códigos de longitud fija como el $C1$ trivialmente satisfacen la condición de prefijos.

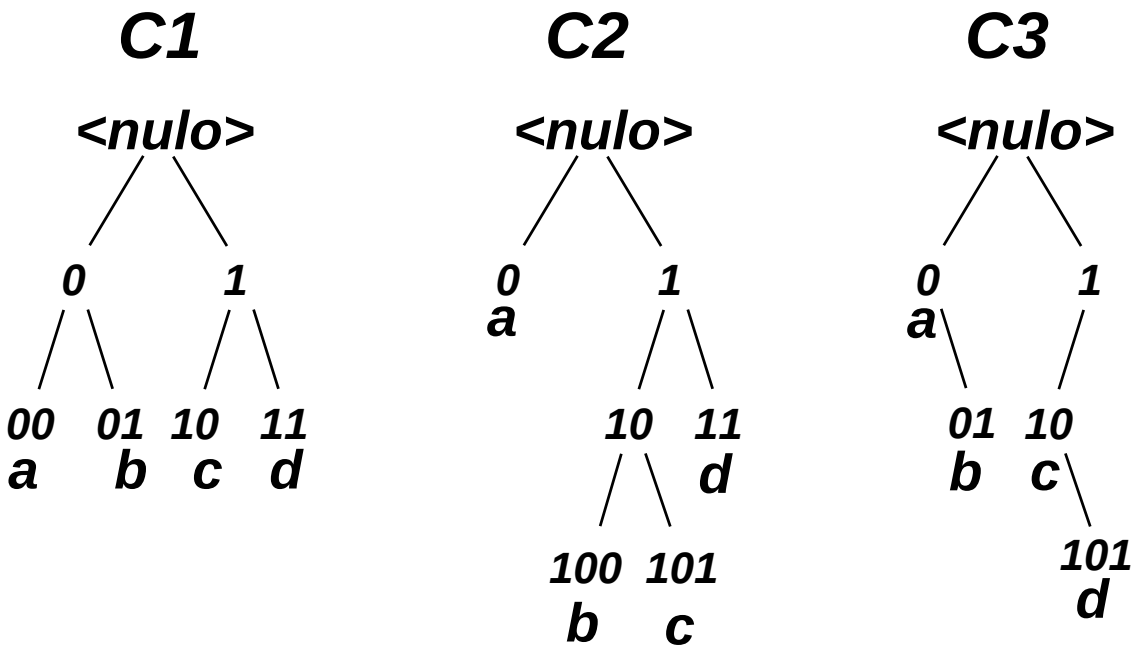
Representación con árboles de Huffman



En esta representación, asociamos con cada nodo de un AB un código binario de la siguiente manera. Al nodo raíz lo asociamos con el código de longitud 0 para el resto de los nodos la definición es recursiva: si un nodo tiene código $b_0 b_1 ... b_{n-1}$ entonces el hijo izquierdo tiene código $b_0 b_1 ... b_{n-1} 0$ y el hijo derecho $b_0 b_1 ... b_{n-1} 1$. Así se van generando todos los códigos binarios posibles. Notar que en el nivel l se encuentran todos los códigos de longitud l.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

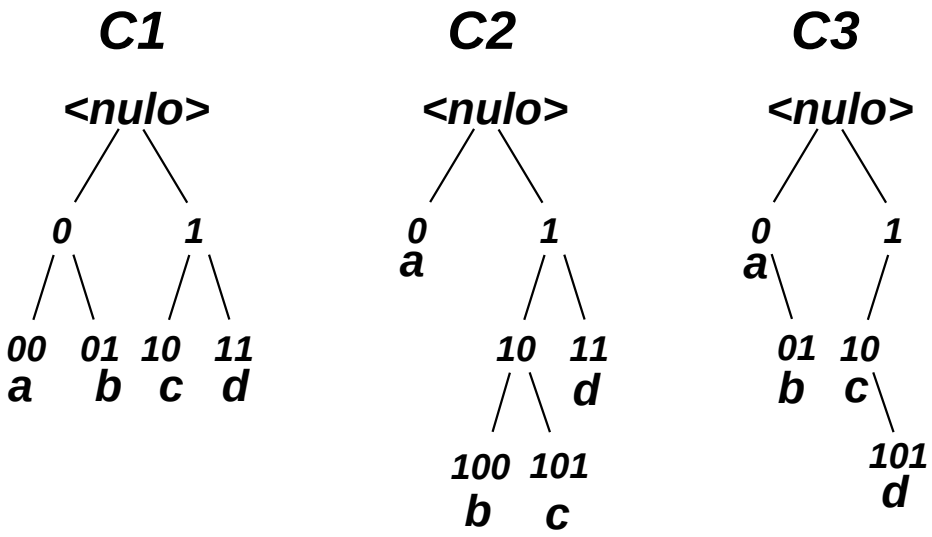
Representación con árboles de Huffman (cont.)



Un código se puede representar como un AB marcando los nodos correspondientes a los códigos de cada uno de los caracteres y eliminando todos los nodos que no están contenidos dentro de los caminos que van desde esos nodos a la raíz. La longitud del código de un caracter es la profundidad del nodo correspondiente.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Representación con árboles de Huffman (cont.)

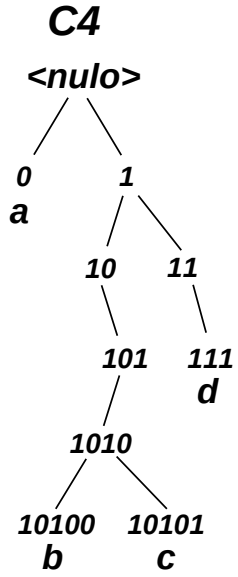


Una forma visual de construir el árbol es comenzar con un árbol vacío y comenzar a dibujar los caminos correspondientes al código de cada caracter. Por construcción, a cada una de las hojas del árbol le corresponde un caracter. Notemos que en el caso del código $C3$ el camino del caracter a esta contenido en el de b y el de c está contenido en el de d. La condición de prefijos equivale a que los caracteres estén *sólo en las hojas*, es decir *no en los nodos interiores*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Códigos redundantes

Letra	Código $C2$	Código $C4$
a	**0**	**0**
b	**100**	**10100**
c	**101**	**10101**
d	**11**	**111**



Notemos que el árbol tiene 3 nodos interiores 10, 101 y 11 que *tienen un sólo hijo*. Entonces podemos eliminar tales nodos interiores, *"subiendo"* todo el subárbol de 1010 a la posición del 10 y la del 111 a la posición del 11. El código resultante resulta ser igual al $C2$ que ya hemos visto.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Códigos redundantes (cont.)

Como todos los nodos suben y la profundidad de los nodos da la longitud del código, es obvio que la longitud del código medio será siempre menor para el código $C2$ que para el $C4$, *independientemente de las probabilidades de los códigos de cada caracter*. Decimos entonces que un código como el $C4$ que tiene nodos interiores con un sólo hijo es *"redundante"*, ya que puede ser trivialmente optimizado eliminando tales nodos y subiendo sus subárboles. Si un AB es tal que no contiene nodos interiores con un solo hijo se le llama *"árbol binario completo"* (ABC). Es decir, en un árbol binario completo los nodos son o bien hojas, o bien nodos interiores con sus dos hijos.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de todos los posibles árboles

Dados una serie de árboles $T_0, T_1, \ldots, T_{n-1}$, llamaremos $\mathrm{comb}(T_0, T_1, \ldots, T_{n-1})$ a la lista de todos los posibles árboles formados combinando $T_0, \ldots, T_{n-1}$. Para $n = 2$ hay una sola combinación que consiste en poner a T_0 como hijo izquierdo y T_1 como hijo derecho de un nuevo árbol. Para aplicar el algoritmo de búsqueda exhaustiva debemos poder generar la lista de árboles $\mathrm{comb}(T_0, T_1, \ldots, T_{n_{\mathrm{char}}-1})$ donde T_j es un árbol que contiene un sólo nodo con el caracter j-ésimo.

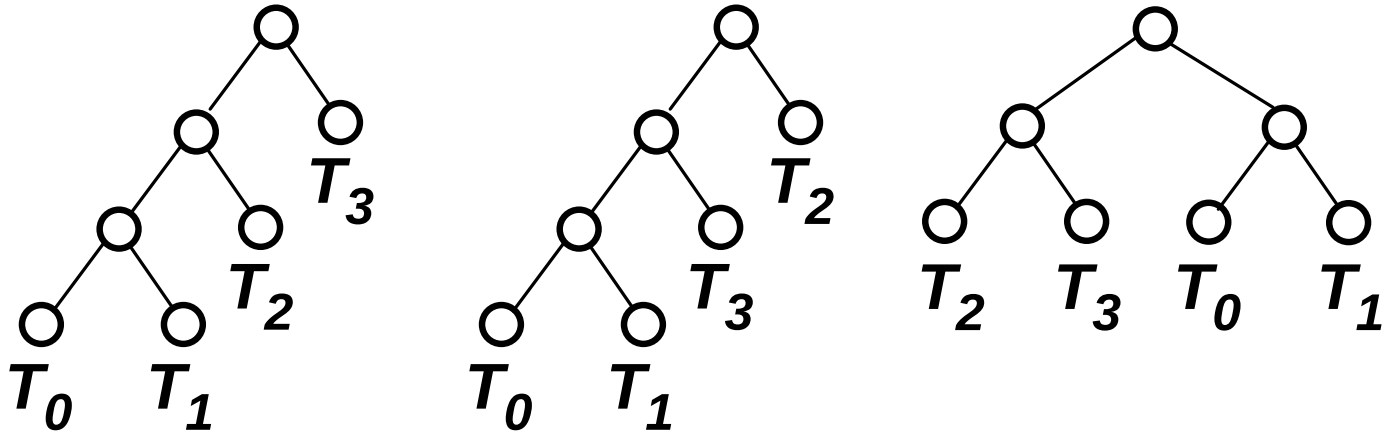
Generación de todos los posibles árboles (cont.)

Todas las combinaciones se pueden hallar tomando cada uno de los posibles pares de árboles (T_i, T_j) de la lista, combinándolo e insertando $\mathrm{comb}(T_i, T_j)$ en la lista, después de eliminar los árboles originales.

$$\begin{aligned}\mathrm{comb}(T_0, T_1, T_2, T_3) = (&\mathrm{comb}(\mathrm{comb}(T_0, T_1), T_2, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_0, T_2), T_1, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_0, T_3), T_1, T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_1, T_2), T_0, T_3),\\ &\mathrm{comb}(\mathrm{comb}(T_1, T_3), T_0, T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_2, T_3), T_0, T_1))\end{aligned}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de todos los posibles árboles (cont.)



Ahora, recursivamente, cada uno de las sublistas se expande a su vez en 3 árboles, por ejemplo

$$\begin{aligned}\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_2,T_3) = (&\mathrm{comb}(\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_2),T_3),\\ &\mathrm{comb}(\mathrm{comb}(\mathrm{comb}(T_0,T_1),T_3),T_2),\\ &\mathrm{comb}(\mathrm{comb}(T_2,T_3),\mathrm{comb}(T_0,T_1)))\end{aligned}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Generación de todos los posibles árboles (cont.)

Entonces, cada una de las sublistas genera 3 árboles. En general vamos a tener

$N_{\mathrm{abc}}(n) =$ **(número de posibles pares a escoger de** n **árboles)** $\times N_{\mathrm{abc}}(n-1)$

donde $N_{\mathrm{abc}}(n)$ **es el número de árboles binarios completos de** n **hojas. El número de posibles pares a escoger de** n **árboles es** $n(n-1)/2$**. La división por 2 proviene de que no importa el orden entre los elementos del par.**

Generación de todos los posibles árboles (cont.)

Aplicando recursivamente esta relación llegamos a

$$N_{\mathrm{abc}}(n) = \frac{n(n-1)}{2}\,\frac{(n-1)(n-2)}{2}\cdots\frac{3\cdot 2}{2}$$
$$= \frac{n!(n-1)!}{2^{n+1}} = \frac{(n!)^2}{n\,2^{n+1}}$$

Usando la aproximación de Stirling llegamos a

$$N_{\mathrm{abc}}(n) = O\left(\frac{n^{2n-1}}{2^n}\right)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El algoritmo de Huffman

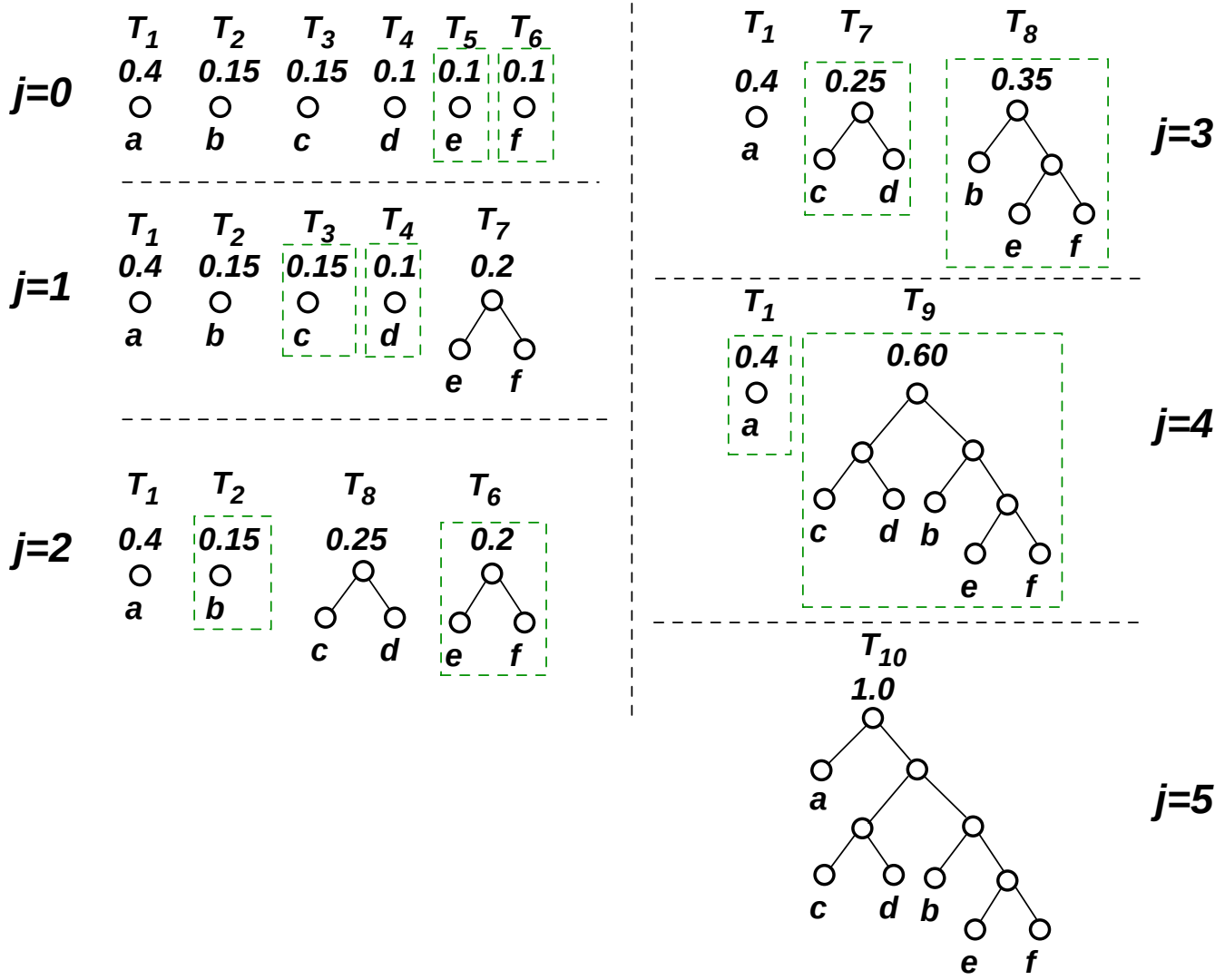
Un algoritmo que permite obtener tablas de código óptimas en tiempos reducidos es el *"algoritmo de Huffman"*. Notemos que es deseable que los caracteres con mayor probabilidad (los más *"pesados"*) deberían estar cerca de la raíz, para tener un código lo más corto posible. Ahora bien, para que un caracter pesado pueda *"subir"*, es necesario que otros caracteres (los más livianos) *"bajen"*. De esta manera podemos pensar que hay una competencia entre los caracteres que tratan de estar lo más cerca posible de la raíz, pero tienden a *"ganar"* los más pesados. Caracteres con probabilidades similares deberían tender a estar en niveles parecidos. Esto sugiere ir apareando caracteres livianos con pesos parecidos en árboles que pasan a ser caracteres *"comodines"* que representan a un conjunto de caracteres.

El algoritmo de Huffman (cont.)

Consigna: **Dados los caracteres a, b, c, d, e, f con pesos $P(a) = 0.4$, $P(b) = 0.15$, $P(c) = 0.15$, $P(d) = 0.1$, $P(e) = 0.1$, $P(f) = 0.1$, encontrar una tabla óptima de codificación, usando el algoritmo de Huffman.**

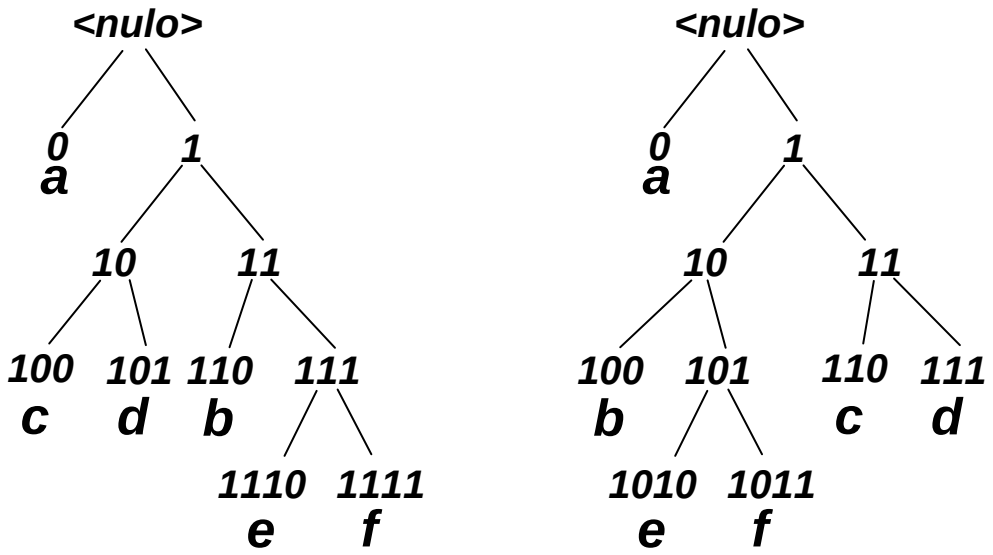
El algoritmo procede de la siguiente manera,

1. **Inicialmente se crean n_c árboles (tantos como caracteres hay). Los árboles tienen una sola hoja asociada con cada uno de los caracteres. Al árbol se le asocia un peso total, inicialmente cada árbol tiene el peso del caracter correspondiente.**
2. **En sucesivas iteraciones el algoritmo va combinando los dos árboles con menor peso en uno sólo. Como en cada combinación desaparecen dos árboles y aparece uno nuevo en $n_c - 1$ iteraciones queda un sólo árbol con el código resultante.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El algoritmo de Huffman (cont.)



La longitud media del código es,

$$\langle l \rangle = 0.4 \times 1 + 0.15 \times 3 + 0.15 \times 3 + 0.1 \times 3 + 0.1 \times 5 + 0.1 \times 4 = 2.5 \,\text{bits/caracter}.$$

resultando en una ganancia del 17% con respecto a la longitud media de 3 que correspondería a la codificación con códigos de igual longitud.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El algoritmo de Huffman (cont.)

Notemos que al combinar dos árboles entre sí no estamos especificando cuál queda como hijo derecho y cuál como izquierdo. Por ejemplo, si al combinar T_7 con T_8 para formar T_9 dejamos a T_7 a la derecha, entonces el árbol resultante será el de la derecha. Si bien el árbol resultante (y por lo tanto la tabla de códigos) es diferente, *la longitud media del código será la misma*, ya que la profundidad de cada caracter, que es su longitud de código, es la misma en los dos árboles. Por ejemplo, e y f tienen longitud 4 en los dos códigos.

A diferencia de los algoritmos heurísticos, la tabla de códigos así generada es *"óptima"* (la mejor de todas), y la longitud media por caracter coincide con la que se obtendría aplicando el algoritmo de búsqueda exhaustiva pero a un costo mucho menor.

Implementación del algoritmo

```
1  struct huffman_tree {
2    double p;
3    btree<int> T;
4  };
5
6  void
7  huffman(const vector<double> &prob,btree<int> &T) {
8    typedef list<huffman_tree> bosque_t;
9
10   // Contiene todos los arboles
11   bosque_t bosque;
12   // Numero de caracteres del codigo
13   int N = prob.size();
14   // Crear los arboles iniciales poniendolos en
15   // una lista Los elementos de la lista contienen
16   // la probabilidad de cada caracter y un arbol
17   // con un solo nodo. Los nodos interiores del
18   // arbol tienen un -1 (es solo para
19   // consistencia) y las hojas tienen el indice
20   // del caracter. (entre 0 y N-1)
21   for (int j=0; j<N; j++) {
22     // Agrega un nuevo elemento a la lista
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23      bosque_t::iterator htree =
24        bosque.insert(bosque.begin(),huffman_tree());
25      htree->p = prob[j];
26      htree->T.insert(htree->T.begin(),j);
27    }
28
29    // Aqui empieza el algoritmo de Huffman.
30    // Tmp va a contener el arbol combinado
31    btree<int> Tmp;
32    for (int j=0; j<N-1; j++) {
33      // En la raiz de Tmp (que es un nodo interior)
34      // ponemos un -1 (esto es solo para chequear).
35      Tmp.insert(Tmp.begin(),-1);
36      // Tmp_p es la probabilidad del arbol combinado
37      // (la suma de las probabilidades de los dos subarboles)
38      double Tmp_p = 0.0;
39      // Para 'k=0' toma el menor y lo pone en el
40      // hijo izquierdo de la raiz de Tmp. Para 'k=1' en el
41      // hijo derecho.
42      for (int k=0; k<2; k++) {
43        // recorre el 'bosque' (la lista de arboles)
44        // busca el menor. 'qmin' es un iterator al menor
45        bosque_t::iterator q = bosque.begin(), qmin=q;
46        while (q != bosque.end()) {
47          if (q->p < qmin->p) qmin = q;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
48          q++;
49        }
50        // Asigna a 'node' el hijo derecho o izquierdo
51        // de la raiz de 'Tmp' dependiendo de 'k'
52        btree<int>::iterator node = Tmp.begin();
53        node = (k==0 ? node.left() : node.right());
54        // Mueve todo el nodo que esta en 'qmin'
55        // al nodo correspondiente de 'Tmp'
56        Tmp.splice(node,qmin->T.begin());
57        // Acumula las probabilidades
58        Tmp_p += qmin->p;
59        // Elimina el arbol correspondiente del bosque.
60        bosque.erase(qmin);
61      }
62      // Inserta el arbol combinado en el bosque
63      bosque_t::iterator r =
64        bosque.insert(bosque.begin(),huffman_tree());
65      // Mueve todo el arbol de 'Tmp' al nodo
66      // recien insertado
67      r->T.splice(r->T.begin(),Tmp.begin());
68      // Pone la probabilidad en el elemento de la
69      // lista
70      r->p = Tmp_p;
71    }
72    // Debe haber quedado 1 solo elemento en la lista
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
73    assert(bosque.size()==1);
74    // Mueve todo el arbol que quedo a 'T'
75    T.clear();
76    T.splice(T.begin(),bosque.begin()->T.begin());
77  }
```

- El código se basa en usar una lista de estructuras de tipo `huffman_tree` que contienen un doble (la probabilidad) y un árbol.
- Los árboles son de tipo `btree<int>`. En los valores nodales almacenaremos para las hojas un índice que identifica al caracter correspondiente. Este índice va entre 0 y `N-1` donde `N` es el número de caracteres. `prob` es un vector de dobles de longitud `N`. `prob[j]` es la probabilidad del caracter `j`. En los valores nodales de los nodos interiores del árbol almacenaremos un valor `-1`. Este valor no es usado normalmente, sólo sirve como un chequeo adicional.
- El tipo `bosque_t` es un alias para una lista de tales estructuras.
- La función `huffman(prob,T)` toma un vector de dobles (las probabilidades) `prob` y calcula el árbol de Huffman correspondiente.
- En el primer lazo inicial sobre `j` los elementos del bosque son inicializados, insertando el único nodo
- En el segundo lazo sobre `j` se van tomando los dos árboles de `bosque` con probabilidad menor. Se combinan usando `splice` en un árbol auxiliar

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

`Tmp`. Los dos árboles son eliminados de la lista y el combinado con la suma de las probabilidades es insertado en el bosque.

- Inicialmente `Tmp` está vacío, y dentro del lazo, la última operación es hacer un `splice` de todo `Tmp` a uno de los árboles del bosque, de manera que está garantizado que al empezar el lazo `Tmp` ***siempre*** está vacío.
- Primero insertamos en `Tmp` un nodo interior (con valor -1). Los dos árboles con pesos menores quedarán como hijos de este nodo raíz.
- La variable `Tmp_p` es la probabilidad combinada. Al empezar el cuerpo del lazo es inicializada a 0 y luego le vamos acumulando las probabilidades de cada uno de los dos árboles a combinar.
- Para evitar la duplicación de código hacemos la búsqueda de los dos menores dentro del lazo sobre `k`, que se hace para `k=0` y `k=1`. Para `k=0` se encuentra el árbol del bosque con menor probabilidad y se inserta en el subárbol izquierdo de `Tmp`. Para `k=1` se inserta al segundo menor en el hijo derecho de de `Tmp`.
- La búsqueda del menor se hace en el lazo sobre `q`, que es un iterator a `list<huffman_tree>` de manera que `*q` es de tipo `huffman_tree` y la probabilidad correspondiente está en `(*q).p` o, lo que es lo mismo `q->p`. Vamos guardando la posición en la lista del mínimo actual en la posición `qmin`.
- Después de encontrar el menor hacemos que el iterator `node` apunte

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

primero al nodo raíz de `Tmp` y después al hijo izquierdo o derecho, dependiendo de `k`.

- El árbol con menor probabilidad es pasado del elemento del bosque al subárbol correspondiente de `Tmp` con splice. Recordar que, como el elemento de la lista es de tipo `huffman_tree`, el árbol está en `qmin->T`. El elemento de la lista es borrado con `erase()` (de lista).
- Las probabilidades de los dos subárboles se van acumulando en `Tmp_p`
- Después de salir del lazo debe quedar en el bosque un solo árbol. `bosque.begin()` es un iterator al primer (y único) elemento del bosque. De manera que el árbol está en `bosque.begin()->T`. La probabilidad correspondiente debería ser 1. El `splice` mueve todo este subárbol al valor de retorno, el árbol `T`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Un programa de compresión de archivos

En el apunte se presenta un programa que comprime archivos utilizando el algoritmo de Huffman. El programa de compresión aquí puede comprimir y descomprimir archivos de texto y no formateados. Para un archivo de texto típico la tasa de compresión es del orden del 35%. Esto es poco comparado con los compresores usuales como `gzip`, `zip` o `bzip2` que presentan tasas de compresión en el orden del 85%. Además, esos algoritmos comprimen los archivos *"al vuelo"* (*"on the fly"*) es decir que no es necesario tener todo el archivo a comprimir en memoria o realizar dos pasadas sobre el mismo.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Conjuntos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Conjuntos

- **Un conjunto es una colección de *"miembros"* o *"elementos"* de un *"conjunto universal"*.**
- **Por contraposición con las listas y otros contenedores vistos previamente, todos los miembros de un conjunto deben ser diferentes, es decir no puede haber dos copias del mismo elemento.**
- **Si bien para definir el concepto de conjunto sólo es necesario el concepto de igualdad o desigualdad entre los elementos del conjunto universal, en general las representaciones de conjuntos asumen que entre ellos existe además una *"relación de orden estricta"*, que usualmente se denota como $<$.**
- **A veces un tal orden no existe en forma natural y es necesario saber definirlo, aunque sea sólo para implementar el tipo conjunto**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Relaciones de orden

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Definición de una relación de orden

[Ojo! esto está en el cap. 2.]

Para los tipos númericos básicos se puede usar el orden usual y para las cadenas de caracteres el orden lexicográfico (alfabético). Para otros conjuntos universales como por ejemplo el conjunto de los pares de enteros la definición de una tal relación de orden puede no ser trivial. Sería natural asumir que

$$(2,3) < (5,6)$$

ya que cada uno de las componentes del primer par es menor que la del segundo par, pero no sabríamos como comparar $(2,3)$ con $(5,1)$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Definición de una relación de orden (cont.)

Primero definamos más precisamente qué es una *"relación de orden"*.
Definición: *"$<$" es una relación de orden en el conjunto C si,*

1. *$<$ es transitiva, es decir, si $a < b$ y $b < c$, entonces $a < c$.*
2. *Dados dos elementos cualquiera de C, una y sólo una de las siguientes afirmaciones es válida:*
   - $a < b$,
   - $b < a$
   - $a = b$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Definición de una relación de orden (cont.)

Una posibilidad sería comparar ciertas funciones escalares del par, como la suma, o la suma de los cuadrados. Por ejemplo definir que $(a, b) < (c, d)$ si y sólo sí $(a + b) < (c + d)$. Una tal definición satisface transitividad, pero no la otra condición, ya que por ejemplo los pares $(2, 3)$ y $(1, 4)$ no satisfacen ninguna de las tres condiciones.

Notar que una vez que se define un operador $<$, los operadores $\leq$, $>$ y $\geq$, e incluso $=$ se pueden definir fácilmente en términos de $<$.

$a > b$ **si y sólo si** $b < a$

$a \leq b$ **si y sólo si no es** $b < a$

$a = b$ **si y sólo si no es** $b < a$ **o** $a < b$

Definición de una relación de orden (cont.)

Una posibilidad para el conjunto de pares de enteros es la siguiente $(a, b) < (c, d)$ **si** $a < c$ **o** $a = c$ **y** $b < d$**. Notar, que esta definición es equivalente a la definición lexicográfica para pares de letras si usamos el orden alfabético para comparar las letras individuales.**

Probemos ahora la transitividad de esta relación. Sean $(a, b) < (c, d)$ **y** $(c, d) < (e, f)$**, entonces hay cuatro posibilidades**

- $a < c < e$
- $a < c = e$ **y** $d < f$
- $a = c < e$ **y** $b < d$
- $a = c = e$ **y** $b < d < f$

y es obvio que en cada una de ellas resulta ser $(a, b) < (e, f)$**. También es fácil demostrar la otra condición.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Definición de una relación de orden (cont.)

Notar que esta relación de orden puede extenderse a cualquier conjunto universal compuesto de pares de conjuntos los cuales individualmente tienen una relación de orden, por ejemplo pares de la forma *(doble,entero)* o *(entero,string)*. A su vez, aplicando recursivamente el razonamiento podemos ordenar n-tuplas de elementos que pertenezcan cada uno de ellos a conjuntos ordenados. Por lo tanto se puede aplicar a listas y conjuntos de elementos cuyo conjunto universal es ordenado.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación de conjuntos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación de conjuntos

Normalmente escribimos un conjunto enumerando sus elementos entre llaves, por ejemplo $\{1, 4\}$. Debemos recordar que no es lo mismo que una lista, ya que, a pesar de que los enumeramos en forma lineal, *no existe un orden preestablecido entre los miembros de un conjunto*. A veces representamos conjuntos a través de una condición sobre los miembros del conjunto universal, por ejemplo

$$A = \{x \text{ entero } / x \text{ es par } \}$$

De esta forma se pueden definir conjuntos con un número infinito de miembros.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación de conjuntos (cont.)

La principal relación en los conjuntos es la de *"pertenencia"* $\in$, esto es $x \in A$ si x es un miembro de A. Existe un conjunto especial $\emptyset$ llamado el *"conjunto vacío"*. Decimos que A esta incluido en B ($A \subseteq B$, o $B \supseteq A$) si todo miembro de A también es miembro de B. También decimos que A es un *"subconjunto"* de B y que B es un *"supraconjunto"*de A. Todo conjunto está incluido en sí mismo y el conjunto vacío está incluido en cualquier conjunto. A y B son *"iguales"* si $A \subseteq B$ y $B \subseteq A$, por lo tanto dos conjuntos son distintos si al menos existe un elemento de A que no pertence a B o viceversa. El conjunto A es un *"subconjunto propio"* (*"supraconjunto propio"*)de B si $A \subseteq B$ ($A \supseteq B$) y $A \neq B$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Notación de conjuntos (cont.)

Las operaciones más básicas de los conjuntos son la *"unión"*, *"intersección"* y *"diferencia"*. La unión $A \cup B$ es el conjunto de los elementos que pertenecen a A *o* a B mientras que la intersección $A \cap B$ es el de los elementos que pertenecen a A *y* a B. Dos conjuntos son *"disjuntos"* si $A \cap B = \emptyset$. La diferencia $A - B$ está formada por los elementos de A que no están en B. Es fácil demostrar la siguiente igualdad de conjuntos

$$A \cup B = (A \cap B) \cup (A - B) \cup (B - A)$$

siendo los tres conjuntos del miembro derecho disjuntos.

Interfase básica para conjuntos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para conjuntos

```
1  class iterator_t {
2  private:
3    /* . . . */;
4  public:
5    bool operator!=(iterator_t q);
6    bool operator==(iterator_t q);
7  };
```

- Como en los otros contenedores STL vistos, una clase `iterator` permite recorrer el contenedor. Los iterators soportan los operadores de comparación `==` y `!=`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para conjuntos (cont.)

```
1  class set {
2  private:
3    /* . . . */;
4  public:
5    set();
6    set(const set &);
7    ~set();
8    elem_t retrieve(iterator_t p);
9    pair<iterator_t,bool> insert(elem_t t);
```

- Sin embargo, en el conjunto no se puede insertar un elemento en una posición determinada, por lo tanto la función `insert` no tiene un argumento posición como en listas o árboles. Sin embargo `insert` retorna un iterator al elemento insertado.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para conjuntos (cont.)

```
1    void erase(iterator_t p);
2    int erase(elem_t x);
```

- La función `erase(p)` elimina el elemento que está en la posición `p`. La posición `p` debe ser válida y dereferenciable, es decir debe haber sido obtenida de un `insert(x)` o `find(x)`. `erase(p)` invalida `p` y todas las otras posiciones obtenidas previamente.
- `erase(x)` elimina el elemento `x` si estaba en el conjunto. Si no, el conjunto queda inalterado.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para conjuntos (cont.)

```
1     void clear();
2     iterator_t next(iterator_t p);
3     iterator_t find(elem_t x);
4     iterator_t begin();
5     iterator_t end();
6   };
```

- Como es usual `begin()`, `end()` y `next()` permiten iterar sobre el conjunto.
- `p=find(x)` devuelve un iterator a la posición ocupada por el elemento `x` en el conjunto. Si el conjunto no contiene a `x` entonces devuelve `end()`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase básica para conjuntos (cont.)

```
1  void set_union(set &A,set &B,set &C);
2  void set_intersection(set &A,set &B,set &C);
3  void set_difference(set &A,set &B,set &C);
```

- Las operaciones binarias sobre conjuntos se realizan con las funciones `set_union(A,B,C)`, `set_intersection(A,B,C)` y `set_difference(A,B,C)` que corresponden a las operaciones $C = A \cup B$, $C = A \cap B$ y $C = A - B$, respectivamente. Notar que estas *no son miembros de la clase*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos

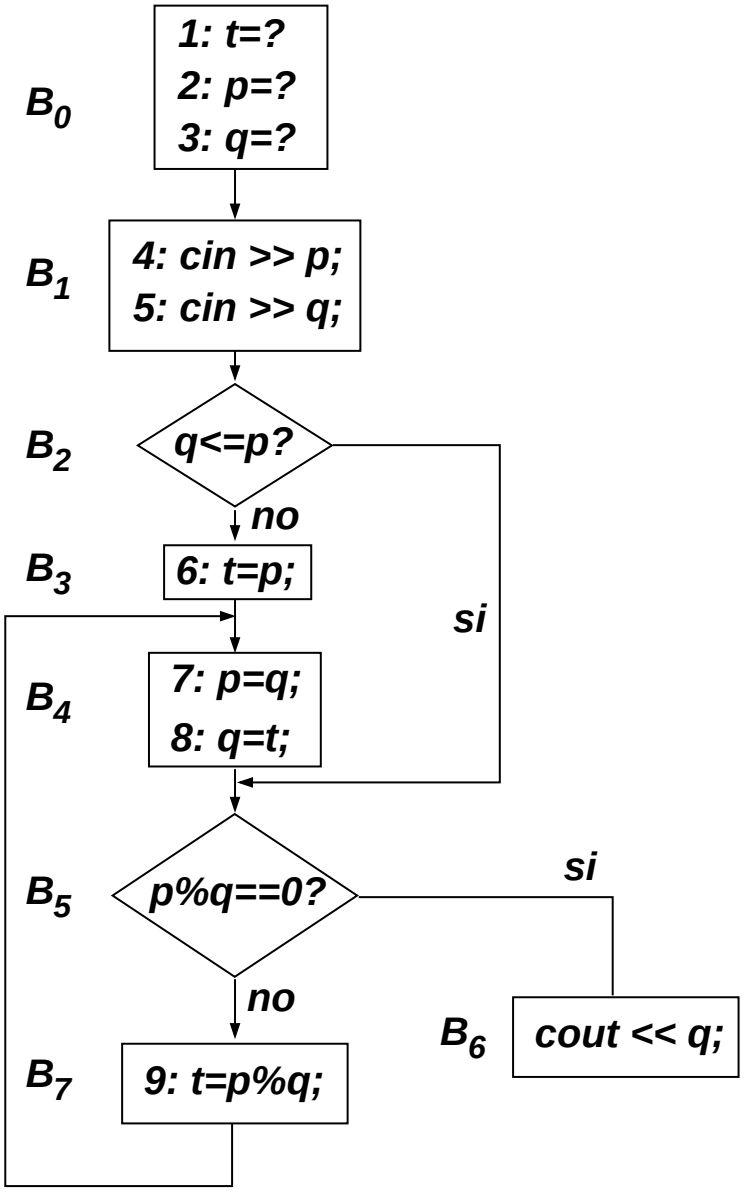
Consideremos un programa simple que calcula el máximo común divisor $\gcd(p, q)$ **de dos números enteros** p**,** q**, mediante el algoritmo de Euclides. Recordemos que el algoritmo de Euclides se basa en la relación recursiva**

$$\gcd(p, q) = \begin{cases} q \text{ divide a } p: & q; \\ \text{si no:} & \gcd(q, \text{rem}(p, q)) \end{cases}$$

donde asumimos que $p > q$ **y** $\text{rem}(p, q)$ **es el resto de dividir** p **por** q**. Por ejemplo, si** $p = 30$ **y** $q = 12$ **entonces**

$$\gcd(30, 12) = \gcd(12, 6) = 6$$

ya que $\text{rem}(30, 12) = 6$ **y 6 divide a 12.**



gen[0]={1,2,3}
kill[0]={4,5,6,7,8,9}

gen[1]={4,5}
kill[1]={2,3,7,8}

gen[2]=kill[2]={}

gen[3]={6}
kill[3]={1,9}

gen[4]={7,8}
kill[4]={2,3,4,5}

gen[5]=kill[5]={}

gen[6]=kill[6]={}

gen[7]={9}
kill[7]={1,6}

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

- **Los bloques B_4, B_5 y B_7 son la base recursiva del algoritmo. La lectura de los datos se produce en el bloque B_1 y el condicional del bloque B_2 se encarga de intercambiar p y q en el caso de que $q > p$.**
- **Los bloques representan porciones de código en los cuales el código sucede secuencialmente línea a línea. Los condicionales en los bloques B_2 y B_5 y el lazo que vuelve del bloque B_7 al B_4 rompen la secuencialidad de este código.**

Análisis de flujo de datos (cont.)

- **En el diseño de compiladores es de importancia saber cuál es la última línea donde una variable puede haber tomado un valor al llegar a otra determinada línea. Por ejemplo, al llegar a la línea 7, `t` puede haber tomado su valor de una asignación en la línea 6 o en la línea 9. Este tipo de análisis es de utilidad para la optimización del código. Por ejemplo si al llegar a un cierto bloque sabemos que una variable `x` sólo puede haber tomado su valor de una asignación constante como `x=20;`, entonces en esa línea se puede reemplazar el valor de `x` por el valor 20. También puede servir para la detección de errores. Hemos introducido un bloque ficticio B_0 que asigna valores indefinidos (representados por el símbolo "?"). Si alguna de estas asignaciones están activas al llegar a una línea donde la variable es usada, entonces puede ser que se este usando una variable indefinida. Este tipo de análisis es estándar en la mayoría de los compiladores.**

Análisis de flujo de datos (cont.)

Para cada bloque B_j vamos a tener definidos 4 conjuntos a saber

- `gen[j]`: las asignaciones que son generadas en el bloque B_j. Por ejemplo en el bloque B_1 se generan las asignaciones 4 y 5 para las variables `p` y `q`.
- `kill[j]`: las asignaciones que son eliminadas en el bloque. Por ejemplo, al asignar valores a las variables `p` y `q` en el bloque B_1 cualquier asignación a esas variables que llegue al bloque será eliminada, como por ejemplo, las asignaciones 2 y 3 del bloque ficticio B_0. En este caso podemos detectar fácilmente cuáles son las asignaciones eliminadas pero en general esto puede ser más complejo, de manera que, conservativamente, introducimos en `kill[j]` todas las asignaciones a variables cuyo valor es reasignado en B_j.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

- En el caso del bloque B_1 las variables reasignadas son `p` y `q`, las cuales sólo tienen asignaciones en las líneas 2,3,7 y 8 (sin contas las proprias asginaciones en el bloque). De manera que `kill[1]={2,3,7,8}`. Notar que, por construcción `gen[j]` y `kill[j]` son conjuntos disjuntos.
- `defin[j]` el conjunto total de definiciones que llegan al bloque B_j.
- `defout[j]` el conjunto total de definiciones que salen del bloque B_j ya sea porque son generadas en el bloque, o porque pasan a través del bloque sin sufrir reasignación.

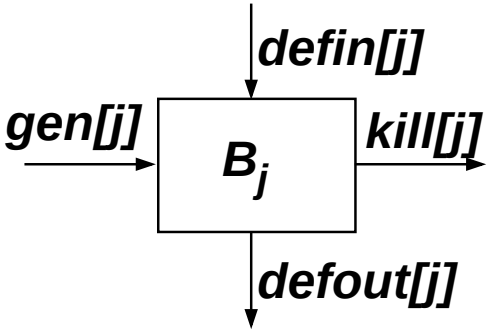
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

En este análisis los conjuntos `gen[j]` y `kill[j]` pueden por simple observación del código, mientras que los `defin[j]` y `defout[j]` son el resultado buscado. *"Ecuación de balance de asignaciones"*:

$$\texttt{defout[j]} = (\texttt{defin[j]} \cup \texttt{gen[j]}) - \texttt{kill[j]}$$

***Las asignaciones que salen del bloque son aquellas que llegan, más las generadas en el bloque menos las que son eliminadas en el mismo*.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

Ahora consideremos las asignaciones que llegan al B_4, es decir `defin[4]`. Estas pueden proceder o bien del bloque B_3 o bien del B_7, es decir

$$\texttt{defin[4]} = \texttt{defout[3]} \cup \texttt{defout[7]}$$

En general tenemos que

$$\texttt{defin[j]} = \bigcup_{\texttt{m}\in\texttt{ent[j]}} \texttt{defout[m]}$$

donde `ent[j]` es el conjunto de bloques cuyas salidas confluyen a la entrada de B_j.

Análisis de flujo de datos (cont.)

En este caso tenemos

$$ent[0] = \emptyset$$

$$ent[1] = \{0\}$$

$$ent[2] = \{1\}$$

$$ent[3] = \{2\}$$

$$ent[4] = \{3, 7\}$$

$$ent[5] = \{2, 4\}$$

$$ent[6] = \{5\}$$

$$ent[7] = \{5\}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
1  void dataflow(vector<set> &gen,
2                vector<set> &kill,
3                vector<set> &defin,
4                vector<set> &defout,
5                vector<set> &ent) {
6    int nblock = gen.size();
7    bool cambio=true;
8    while (cambio) {
9      cambio=false;
10     for (int j=0; j<nblock; j++) {
11       // Calcular la entrada al bloque 'defin[j]'
12       //  sumando sobre los 'defout[m]' que
13       //  confluyen al bloque j . . .
14     }
15     int out_prev = defout[j].size();
16
17     cambio=false;
18     for (int j=0; j<nblock; j++) {
19       // Calcular el nuevo valor de 'defout[j]'
20       //  usando la ec. de balance de asignaciones. . .
21       if (defout[j].size() != out_prev) cambio=true;
22     }
23   }
24 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

La función `dataflow()` toma como argumentos vectores de conjuntos de longitud `nblock` (el número de bloques, en este caso 8). `ent[]`, `gen[]` y `kill[]` son datos de entrada, mientras que `defin[]` y `defout[]` son datos de salida calculados por `dataflow()`. `tmp` es una variable auxiliar de tipo conjunto. El código entra en un lazo infinito en el cual va calculando para cada bloque las asignaciones a la entrada `defin[j]` aplicando la ec. de balance y la union de los `defout[]`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

El proceso es iterativo, de manera que hay que inicializar las variables `defin[]` y `defout[]` y detectar cuando no hay mas cambios. Notemos primero que lo único que importa son las inicializaciones para `defin[]` ya que cualquier valor que tome `defout[]` al ingresar a `dataflow[]` será sobreescrito durante la primera iteración. Tomemos como inicialización $\texttt{defin[j]}^0 = \emptyset$. Después de la primera iteración los `defout[j]` tomarán ciertos valores $\texttt{defout[j]}^0$, posiblemente no nulos, de manera que en la iteración 1 los $\texttt{defin[j]}^1$ pueden eventualmente ser no nulos, pero vale que

$$\texttt{defin[j]}^0 \subseteq \texttt{defin[j]}^1$$

Es fácil ver entonces que, después de aplicar la ec. de balance de asignaciones valdrá que

$$\texttt{defout[j]}^0 \subseteq \texttt{defout[j]}^1$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Análisis de flujo de datos (cont.)

Siguiendo el razonamiento, puede verse que siempre seguirá valiendo que

$$\texttt{defin[j]}^k \subseteq \texttt{defin[j]}^{k+1}$$

$$\texttt{defout[j]}^k \subseteq \texttt{defout[j]}^{k+1}$$

Nos preguntamos ahora cuantas veces hay que ejecutar el algoritmo. Notemos que, como tanto `defin[j]` como `defout[j]` deben ser subconjuntos del conjunto finito que representan todas las asignaciones en el programa, a partir de una cierta iteración los `defin[j]` y `defout[j]` no deben cambiar más. Decimos entonces que el algoritmo *"convirgió"* y podemos detenerlo.

Notar que la ec. previa garantiza que, para detectar la convergencia basta con verificar que el tamaño de ningún `defout[j]` cambie.

```
1  void dataflow(vector<set> &gen,
2                vector<set> &kill,
3                vector<set> &defin,
4                vector<set> &defout,
5                vector<set> &ent) {
6    int nblock = gen.size();
7    set tmp;
8    bool cambio=true;
9    while (cambio) {
10     for (int j=0; j<nblock; j++) {
11       defin[j].clear();
12       iterator_t p = ent[j].begin();
13       while (p!=ent[j].end()) {
14         int k = ent[j].retrieve(p);
15         set_union(defin[j],defout[k],tmp);
16         defin[j] = tmp;
17         p = ent[j].next(p);
18       }
19     }
20     cambio=false;
21     for (int j=0; j<nblock; j++) {
22       int out_prev = defout[j].size();
23       set_union(defin[j],gen[j],tmp);
24       set_difference(tmp,kill[j],defout[j]);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
25          if (defout[j].size()!=out_prev) cambio=true;
26        }
27      }
28    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por vectores de bits

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación por vectores de bits

Tal vez la forma más simple de representar un conjunto es guardando un campo de tipo `bool` por cada elemento del conjunto universal. Si este campo es verdadero entonces el elemento está en el conjunto y viceversa. Por ejemplo, si el conjunto universal son los enteros de `0` a `N-1`

$$U = \{j \text{ entero, tal que } 0 \leq j < N\}$$

entonces podemos representar a los conjuntos por vectores de valores booleanos (puede ser `vector<bool>`) de longitud `N`. Si `v` es el vector, entonces `v[j]` indica si el entero `j` está o no en el conjunto. Por ejemplo, si el conjunto es `S={4,6,9}` y `N` es 10, entonces el vector de bits correspondiente sería `v={0,0,0,0, 1,0,1,0,0,1}`.

Implementación por vectores de bits (cont.)

Para insertar o borrar elementos se prende o apaga el bit correspondiente. Todas estas son operaciones $O(1)$. Las operaciones binarias también son muy simples de implementar. Por ejemplo, si queremos hacer la unión $C = A \cup B$, entonces debemos hacer `C.v[j] = A.v[j] || B.v[j]`. La intersección se obtiene reemplazando `||` con `&&` y la diferencia $C = A - B$ con `C.v[j] = A.v[j] && ! B.v[j]`.

Implementación por vectores de bits (cont.)

Notar que el tiempo de ejecución de estas operaciones es $O(N)$, donde N *es el número de elementos en el conjunto universal.* La memoria requerida es `N` bits, es decir que también es $O(N)$. Es de destacar que también se podría usar `vector<T>` con cualquier tipo `T` convertible a un entero, por ejemplo `int`, `char`. `bool` y sus variantes. En cada caso la memoria requerida es `N*sizeof(T)`, de manera que siempre es $O(N)$. Esto representa 8 bits por elemento para `char` o 32 para `int`. En el caso de `bool`, el operador `sizeof(bool)` reporta normalmente 1 byte por elemento, pero la representación interna de `vector<bool>` y en realidad requiere de un sólo bit por elemento.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Otros conjuntos universales

Para representar conjuntos universales U que no son subconjuntos de los enteros, o que no son un subconjunto contiguo de los enteros `[0,N)` debemos definir funciones que establezcan la correspondencia entre los elementos del conjunto universal y el conjunto de los enteros entre $[0, N)$. Llamaremos a estas funciones

```
1  int indx(elem_t t);
2  elem_t element(int j);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Otros conjuntos universales (cont.)

Funciones para

$$U = \{\textbf{enteros pares entre 100 y 198}\}$$

```
1 const int N=50;
2 typedef int elem_t;
3 int indx(elem_t t) { return (t-100)/2; }
4 elem_t element(int j) { return 100+2*j; }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Otros conjuntos universales (cont.)

Funciones auxiliares para definir conjuntos dentro de las letras *a–z* y *A–Z*.

```
1   const int N=52;
2   typedef char elem_t;
3   int indx(elem_t c) {
4     if (c>='a' && c<='z') return c-'a';
5     else if (c>='A' && c<='Z') return 26+c-'A';
6     else cout << "Elemento fuera de rango!!\n"; abort();
7   }
8   elem_t element(int j) {
9     assert(j<N);
10    return (j<26 ? 'a'+j : 'A'+j-26);
11  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación

```
1   typedef int iterator_t;
2
3   class set {
4   private:
5     vector<bool> v;
6     iterator_t next_aux(iterator_t p) {
7       while (p<N && !v[p]) p++;
8       return p;
9     }
10    typedef pair<iterator_t,bool> pair_t;
11  public:
12    set() : v(N,0) { }
13    set(const set &A) : v(A.v) {}
14    ~set() {}
15    iterator_t lower_bound(elem_t x) {
16      return next_aux(indx(x));
17    }
18    pair_t insert(elem_t x) {
19      iterator_t k = indx(x);
20      bool inserted = !v[k];
21      v[k] = true;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22      return pair_t(k,inserted);
23    }
24    elem_t retrieve(iterator_t p) { return element(p); }
25    void erase(iterator_t p) { v[p]=false; }
26    int erase(elem_t x) {
27      iterator_t p = indx(x);
28      int r = (v[p] ? 1 : 0);
29      v[p] = false;
30      return r;
31    }
32    void clear() { for(int j=0; j<N; j++) v[j]=false; }
33    iterator_t find(elem_t x) {
34      int k = indx(x);
35      return (v[k] ? k : N);
36    }
37    iterator_t begin() { return next_aux(0); }
38    iterator_t end() { return N; }
39    iterator_t next(iterator_t p) {
40      return next_aux(++p);
41    }
42    int size() {
43      int count=0;
44      for (int j=0; j<N; j++) if (v[j]) count++;
45      return count;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
46      }
47      friend void set_union(set &A,set &B,set &C);
48      friend void set_intersection(set &A,set &B,set &C);
49      friend void set_difference(set &A,set &B,set &C);
50    };
51
52    void set_union(set &A,set &B,set &C) {
53      for (int j=0; j<N; j++) C.v[j] = A.v[j] || B.v[j];
54    }
55    void set_intersection(set &A,set &B,set &C) {
56      for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
57    }
58    void set_difference(set &A,set &B,set &C) {
59      for (int j=0; j<N; j++) C.v[j] = A.v[j] && ! B.v[j];
60    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación (cont.)

- El vector de bits está almacenado en un campo `vector<bool>`. El constructor por defecto dimensiona a `v` de tamaño `N` y con valores 0. El constructor por copia copia el campo `v`, con lo cual copia el conjunto.
- La clase iterator es un `typedef` de los enteros, con la particularidad de que el iterator corresponde al índice en el conjunto universal, es decir el valor que retorna la función `indx()`. Por ejemplo, en el caso de los enteros pares entre 100 y 198 tenemos 50 valores posibles del tipo iterator. El iterator correspondiente a 100 es 0, a 102 le corresponde 1, y así siguiendo hasta 198 que le corresponde 50.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación (cont.)

- Se elige como iterator `end()` el índice `N`, ya que este no es nunca usado por un elemento del conjunto.
- Por otra parte, los iterators sólo deben avanzar sobre los valores definidos en el conjunto. Por ejemplo, si el conjunto es `S={120,128,180}` los iterators ocupados son 10, 14 y 40. Entonces `p=S.begin();` debe retornar 10 y aplicando sucesivamente `p=S.next(p);` debemos tener `p`=14, 40 y 50 (que es `N=S.end()`).
- La función auxiliar `p=next_aux(p)` (notar que esta declarada como privada) devuelve el primer índice siguiente a `p` ocupado por un elemento. Por ejemplo en el caso del ejemplo, `next_aux(0)` debe retornar 10, ya que el primer índice ocupado en el conjunto siguiente a 0 es el 10.
- La función `next()` (que en la interfase avanzada será sobrecargada sobre el operador `operator++`) incrementa `p` y luego le aplica `next_aux()` para avanzar hasta el siguiente elemento del conjunto.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación (cont.)

- La función `retrieve(p)` devuelve el elemento usando la función `element(p)`.
- `erase(x)` e `insert(x)` prenden o apagan la posición correspondiente `v[indx(x)]`. Notar que son $O(1)$.
- `find(x)` verifica si el elemento está en el conjunto, en ese caso retorna `indx(x)`, si no retorna `N` (que es `end()`).
- `size()` cuenta las posiciones en `v` que están prendidas. Por lo tanto es $O(N)$.
- Las funciones binarias `set_union(A,B,C)`, `set_intersection(A,B,C)` y `set_difference(A,B,C)` hacen un lazo sobre todas las posiciones del vector y por lo tanto son $O(N)$.

Implementación con listas

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación con listas

- **Podemos representar conjuntos con correspondencias, usando las claves de la correspondencia como elementos del conjunto e ignorando el valor del contradominio.**
- **Las operaciones binarias `set_union(A,B,C)`, `set_intersection(A,B,C)` y `set_difference(A,B,C)` no tiene equivalencia dentro de la correspondencia.**
- **Si tenemos una implementación de CORRESPONDENCIA, sólo debemos implementar las funciones binarias**

Similaridad entre conjunto y correspondencia:

TAD conjunto	TAD Correspondencia
`x = retrieve(p);`	`x = retrieve(p).first;`
`p=insert(x)`	`p=insert(x,w)`
`erase(p)`	`erase(p)`
`erase(x)`	`erase(x)`
`clear()`	`clear()`
`p = find(x)`	`p = find(x)`
`begin()`	`begin()`
`end()`	`end()`

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Similaridad entre conjunto y correspondencia: (cont.)

- **Consideremos la implementación de correspondencia por listas ordenadas y no ordenadas a conjuntos. En ambos casos las operaciones de inserción supresión terminan siendo $O(n)$ ya que hay que recorrer el contenedor para encontrar el elemento.**
- **Sin embargo, hay una gran diferencia para el caso de las operaciones binarias. Consideremos por ejemplo `set_intersection(A,B,C)`. En el caso de contenedores no ordenados, la única posibilidad es comparar cada uno de los elementos x_a de A con cada uno de los elementos de B. Si se encuentra el elemento x_a en B, entonces el elemento es insertado en `C`. Tal algoritmo es $O(n_A\, n_B)$, donde $n_{A,B}$ es el número de elementos en A y B. Si el número de elementos en los dos contenedores es similar ($n_A \sim n_B \sim n$), entonces vemos que es $O(n^2)$.**

Operaciones binarias

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones binarias con listas

Con listas ordenadas se puede implementar una versión de `set_union(A,B,C)` que es $O(n)$. Mantenemos dos posiciones `pa` y `pb`, cuyos valores llamaremos x_a y x_b, tales que

- los elementos en A en posiciones anteriores a `pa` (y por lo tanto menores que x_a) son menores que x_b y viceversa,
- todos los valores en B antes de `pb` son menores que x_a.

Estas condiciones son bastante fuertes: si el valor $x_a < x_b$ entonces podemos asegurar que $x_a \notin B$. Todos los elementos anteriores a x_b son menores a x_a y por lo tanto distintos a x_a. Por otra parte los que están después de x_b son mayores que x_b y por lo tanto que x_a, con lo cual también son distintos. Como conclusión, no hay ningún elemento en B que sea igual a x_a, es decir $x_a \notin B$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones binarias con listas (cont.)

Inicialmente podemos poner `pa=A.begin()` y `pb=B.begin()`. En cada paso puede ser que avancemos `pa`, `pb` o ambos. Por ejemplo, consideremos el caso

$$A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\}$$

Inicialmente tenemos $x_a = 1$ y $x_b = 3$. Si avanzamos `pb`, de manera que x_b pasa a ser 5, entonces la segunda condición no se satisfará, ya que el 3 en B no es menor que x_a que es 1. Por otra parte, si avanzamos `pa`, entonces sí se seguirán satisfaciendo ambas condiciones, y en general esto será siempre así, mientras *avancemos siempre el elemento menor*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones binarias con listas (cont.)

Para cualquiera de las operaciones binarias inicializamos los iterators con `pa=A.begin()` y `pb=B.begin()` y los vamos avanzando con el mecanismo explicado, es decir siempre el menor o los dos cuando son iguales. El proceso se detiene cuando alguno de los iterators llega al final de su conjunto. En cada paso alguno de los iterators avanza, de manera que es claro que en un número finito de pasos alguno de los iterators llegará al fin, de hecho en menos de $n_a + n_b$ pasos. Las posiciones `pa` y `pb` recorren todos los elementos de sus respectivos conjuntos.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones binarias con listas (cont.)

Ahora consideremos la operación de `set_union(A,B,C)`. Debemos asegurarnos de insertar todos los elementos de A y B, pero una sola vez y en forma ordenada. Esto se logra si en cada paso insertamos en el fin de C el elemento menor de x_a y x_b (si son iguales se inserta una sola vez). Efectivamente, si en un momento $x_a < x_b$ entonces, por lo discutido previamente x_a seguramente no está en B y podemos insertarlo en C, ya que en el siguiente paso `pa` avanzará, dejándolo atrás, con lo cual seguramente no lo insertaremos nuevamente. Además en pasos previos x_a no puede haber sido insertado ya que si era el menor `pa` habría avanzado dejándolo atrás y si era el mayor no habría sido insertado.

Operaciones binarias con listas (cont.)

Puede verse que los elementos de C quedan ordenados, ya que de x_a y x_b siempre avanzamos el menor. Una vez que uno de los iterators (digamos `pa`) llegó al final, el *"resto"* del otro se inserta directamente al fin. Ej:
$A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\}$

- x_a**=1,** x_b**=3, inserta 1 en** C
- x_a**=3,** x_b**=3, inserta 3 en** C
- x_a**=5,** x_b**=5, inserta 5 en** C
- x_a**=7,** x_b**=7, inserta 7 en** C
- x_a**=10,** x_b**=9, inserta 9 en** C
- x_a**=10,** x_b**=10, inserta 10 en** C
- `pa` **llega a** `A.end()`
- **Inserta todo el resto de** B **(el elemento 12) en** C**.**

quedando $C = \{1, 3, 5, 7, 9, 10, 12\}$ **que es el resultado correcto.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Operaciones binarias con listas (cont.)

En el caso de `set_intersection(A,B,C)` sólo hay que insertar x_a cuando $x_a = x_b$ y los restos no se insertan.
Ej: $A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\}$

- x_a=1, x_b=3,
- x_a=3, x_b=3, inserta 3
- x_a=5, x_b=5, inserta 5
- x_a=7, x_b=7, inserta 7
- x_a=10, x_b=9,
- x_a=10, x_b=10, inserta 10
- `pa` llega a `A.end()`

Operaciones binarias con listas (cont.)

Para `set_difference(A,B,C)` hay que insertar x_a si $x_a < x_b$ y x_b no se inserta nunca. Al final sólo se inserta el resto de A.
Ej: $A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\}$

- x_a**=1,** x_b**=3, inserta 1**
- x_a**=3,** x_b**=3,**
- x_a**=5,** x_b**=5,**
- x_a**=7,** x_b**=7,**
- x_a**=10,** x_b**=9,**
- x_a**=10,** x_b**=10,**
- `pa` **llega a** `A.end()`

Con estos algoritmos, los tiempos de ejecución son $O(n_A + n_B)$ ($O(n)$ si los tamaños son similares).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación

```
1   typedef int elem_t;
2
3   typedef list<elem_t>::iterator iterator_t;
4
5   class set {
6   private:
7     list<elem_t> L;
8   public:
9     set() {}
10    set(const set &A) : L(A.L) {}
11    ~set() {}
12    elem_t retrieve(iterator_t p) { return *p; }
13    iterator_t lower_bound(elem_t t) {
14      list<elem_t>::iterator p = L.begin();
15      while (p!=L.end() && t>*p) p++;
16      return p;
17    }
18    iterator_t next(iterator_t p) { return ++p; }
19    pair<iterator_t,bool> insert(elem_t x) {
20      pair<iterator_t,bool> q;
21      iterator_t p;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22      p = lower_bound(x);
23      q.second = p==end() || *p!=x;
24      if(q.second) p = L.insert(p,x);
25      q.first = p;
26      return q;
27    }
28    void erase(iterator_t p) { L.erase(p); }
29    void erase(elem_t x) {
30      list<elem_t>::iterator
31        p = lower_bound(x);
32      if (p!=end() && *p==x) L.erase(p);
33    }
34    void clear() { L.clear(); }
35    iterator_t find(elem_t x) {
36      list<elem_t>::iterator
37        p = lower_bound(x);
38      if (p!=end() && *p==x) return p;
39      else return L.end();
40    }
41    iterator_t begin() { return L.begin(); }
42    iterator_t end() { return L.end(); }
43    int size() { return L.size(); }
44    friend void set_union(set &A,set &B,set &C);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
45    friend void set_intersection(set &A,set &B,set &C);
46    friend void set_difference(set &A,set &B,set &C);
47  };
48
49  void set_union(set &A,set &B,set &C) {
50    C.clear();
51    list<elem_t>::iterator pa = A.L.begin(),
52      pb = B.L.begin(), pc = C.L.begin();
53    while (pa!=A.L.end() && pb!=B.L.end()) {
54      if (*pa<*pb) { pc = C.L.insert(pc,*pa); pa++; }
55      else if (*pa>*pb) {pc = C.L.insert(pc,*pb); pb++; }
56      else {pc = C.L.insert(pc,*pa); pa++; pb++; }
57      pc++;
58    }
59    while (pa!=A.L.end()) {
60      pc = C.L.insert(pc,*pa);
61      pa++; pc++;
62    }
63    while (pb!=B.L.end()) {
64      pc = C.L.insert(pc,*pb);
65      pb++; pc++;
66    }
67  }
68  void set_intersection(set &A,set &B,set &C) {
69    C.clear();
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
70    list<elem_t>::iterator pa = A.L.begin(),
71      pb = B.L.begin(), pc = C.L.begin();
72    while (pa!=A.L.end() && pb!=B.L.end()) {
73      if (*pa<*pb) pa++;
74      else if (*pa>*pb) pb++;
75      else { pc=C.L.insert(pc,*pa); pa++; pb++; pc++; }
76    }
77  }
78  // C = A - B
79  void set_difference(set &A,set &B,set &C) {
80    C.clear();
81    list<elem_t>::iterator pa = A.L.begin(),
82      pb = B.L.begin(), pc = C.L.begin();
83    while (pa!=A.L.end() && pb!=B.L.end()) {
84      if (*pa<*pb) { pc=C.L.insert(pc,*pa); pa++; pc++; }
85      else if (*pa>*pb) pb++;
86      else { pa++; pb++; }
87    }
88    while (pa!=A.L.end()) {
89      pc = C.L.insert(pc,*pa);
90      pa++; pc++;
91    }
92  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Descripción de la implementación (cont.)

- **Los métodos de la clase son muy similares a los de correspondencia y no serán explicados nuevamente. Consideremos por ejemplo `p=insert(x)`. La única diferencia con el `insert()` de `map` es que en aquel caso hay que insertar un par que consiste en la clave y el valor de contradominio, mientras que en `set` sólo hay que insertar el elemento.**
- **Las funciones binarias no pertenecen a la clase y el algoritmo utilizado responde a lo descripto previamente. Notar que al final de `set_union()` se insertan *los dos restos* de A y B a la cola de C. Esto se debe a que al llegar a este punto uno de los dos iterators está en `end()` de manera que sólo uno de los lazos se ejecutará efectivamente.**

Tiempos de ejecución

Método	$T(N)$
`retrieve(p), insert(x), erase(x), clear(), find(x), lower_bound(x), set_union(A,B,C), set_intersection(A,B,C), set_difference(A,B,C),`	$O(n)$
`erase(p), begin(), end(),`	$O(1)$

((version aed-2.0.4-10-gdb5d209) (date <unknown>) (processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada para conjuntos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada

- La clase `set` pasa a ser ahora un template, de manera que podremos declarar `set<int>`, `set<double>`.
- La clase `iterator` es ahora una clase anidada dentro de `set`. Externamente se verá como `set<int>::iterator`
- La dereferenciación de posiciones (`x=retrieve(p)`) se reemplaza por `x=*p`. sobrecargando el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos dato o métodos, podemos escribir `p->campo` o `p->f(...)`. Para esto sobrecargamos los operadores `operator*` y `operator->`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Interfase avanzada (cont.)

- Igual que con la interfase básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=` en la clase iterator.
- `erase(x)` retorna el número de elementos efectivamente eliminados.
- `insert(x)` retorna un `pair<iterator,bool>`. El primero es, como siempre, un iterator al elemento insertado. El segundo indica si el elemento realmente fue insertado o ya estaba en el conjunto.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diccionarios

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El diccionario

En algunas aplicaciones puede ser que se necesite un TAD como el conjunto pero sin necesidad de las funciones binarias. A un tal TAD lo llamamos TAD DICCIONARIO. Por supuesto cualquier implementación de conjuntos puede servir como diccionario, por ejemplo con vectores de bits, contenedores lineales ordenados o no. Sin embargo existe una implementación muy eficiente para la cual las inserciones y supresiones son $O(1)$, basada en la estructura *"tabla de dispersión"* (*"hash tables"*). Sin embargo, no es simple implementar en forma eficiente las operaciones binarias para esta implementación, por lo cual no es un buen candidato para conjuntos.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión

La idea esencial es que dividimos el conjunto universal en B *"cubetas"* (*"buckets"* o *"bins"*), de tal manera que, a medida que nuevos elementos son insertados en el diccionario, estos son desviados a la cubeta correspondiente. Por ejemplo, si consideramos diccionarios de cadenas de caracteres en el rango `a-z`, es decir letras minúsculas, entonces podemos dividir al conjunto universal en B=26 cubetas. La primera corresponde a todos los strings que comienzan con `a`, la segunda los que comienzan con `b` y así siguiendo hasta la `z`. En general tendremos una función de dispersión `int b = h(elem_t t)` que nos da el número de cubeta `b` en el cual debe ser almacenado el elemento `t`. En el ejemplo descripto, la función podría implementarse como sigue

```
1  int h(string t) {
2    return t[0]-'a';
3  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión (cont.)

En este caso está garantizado que los números de cubetas devueltos por `h()` están en el rango $[0, B)$. En la práctica, el programador de la clase puede proveer funciones de hash para los tipos más usuales (como `int`, `double`, `string`...) dejando la posibilidad de que el usuario defina la función de hash para otros tipos, o también para los tipos básicos si considera que los que el provee son más eficientes (ya veremos cuáles son los requisitos para una buena función de hash). Asumiremos siempre que el tiempo de ejecución de la función de dispersión es $O(1)$. Para mayor seguridad, asignamos al elemento `t` la cubeta `b=h(t)%B`, de esta forma está siempre garantizado que `b` está en el rango $[0, B)$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión (cont.)

Básicamente, las cubetas son guardadas en un arreglo de cubetas (`vector<elem_t> v(B)`). Para insertar un elemento, calculamos la cubeta usando la función de dispersión y guardamos el elemento en esa cubeta. Para hacer un `find(x)` o `erase(x)`, calculamos la cubeta y verificamos si el elemento está en la cubeta o no. De esta forma tanto las inserciones como las supresiones son $O(1)$. Este costo tan bajo es el interés principal de estas estructuras.

Tablas de dispersión (cont.)

Pero normalmente el número de cubetas es mucho menor que el número de elementos del conjunto universal N (en muchos casos este último es infinito). En el ejemplo de los strings, todos los strings que empiezan con `a` van a la primera cubeta. Si un elemento es insertado y la cubeta correspondiente ya está ocupada decimos que hay una *"colisión"* y no podemos insertar el elemento en la tabla. Por ejemplo, si $B = 10$ y usamos `h(x)=x`, entonces si se insertan los elementos $\{1, 13, 4, 1, 24\}$ entonces los tres primeros elementos van a las cubetas 1, 3 y 4 respectivamente. El cuarto elemento también va a la cubeta 1, la cual ya está ocupada, pero no importa ya que es el *mismo* elemento que está en la cubeta. El problema es al insertar el elemento 24, ya que va a parar a la cubeta 4 la cual ya está ocupada, pero por *otro* elemento (el 4).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión (cont.)

Si el número de elementos en el conjunto n es pequeño con respecto al número de cubetas ($n \ll B$) entonces la probabilidad de que dos elementos vayan a la misma cubeta es pequeña, pero en este caso la memoria requerida (que es el tamaño del vector `v`, es decir $O(B)$) sería mucho mayor que el tamaño del conjunto, con lo cual la utilidad práctica de esta implementación sería muy limitada. De manera que debe definirse una estrategia para *"resolver colisiones"*. Hay al menos dos formas bien conocidas:

- **Usar *"tablas de dispersión abiertas"*.**
- **Usar redispersión en *"tablas de dispersión cerradas"*.**

Tablas de dispersión abiertas

En esta implementación las cubetas no son elementos, sino que son listas (simplemente enlazadas) de elementos, es decir el vector `v` es de tipo `vector< list<elem_t> >`). De esta forma cada cubeta puede contener (teóricamente) infinitos elementos. Los elementos pueden insertarse en las cubetas en forma ordenada o desordenada. La discusión de la eficiencia en este caso es similar a la de correspondencia con contenedores lineales.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión abiertas (cont.)

Discutiremos la implementación del TAD diccionario implementado por tablas de dispersión abiertas con listas desordenadas.

- **La inserción de un elemento `x` pasa por calcular el número de cubeta usando la función de dispersión y revisar la lista (cubeta) correspondiente. Si el elemento está en la lista, entonces no es necesario hacer nada. Si no está, podemos insertar el elemento en cualquier posición, puede ser en `end()`.**
- **El costo de la inserción es, en el peor caso, cuando el elemento no esta en la lista, proporcional al número de elementos en la lista (cubeta).**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión abiertas (cont.)

- **Si tenemos n elementos, el número de elementos por cubeta será, en promedio, n/B. Si el número de cubetas es $B \approx n$, entonces $n/B \approx 1$ y el tiempo de ejecución es $O(1 + n/B)$. El 1 tiene en cuenta acá de que al menos hay que calcular la función de dispersión.**
- **Como el término n/B puede ser menor, e incluso mucho menor, que 1, entonces hay que mantener el término 1, de lo contrario estaríamos diciendo que en ese caso el costo de la función es mucho menor que 1.**
- **En el peor caso, todos los elementos pueden terminar en una sola cubeta, en cuyo caso la inserción sería $O(n)$. Algo similar pasa con `erase()`.**

Detalles de implementación

```
1   typedef int key_t;
2
3   class hash_set;
4   class iterator_t {
5     friend class hash_set;
6   private:
7     int bucket;
8     std::list<key_t>::iterator p;
9     iterator_t(int b,std::list<key_t>::iterator q)
10      : bucket(b), p(q) { }
11  public:
12    bool operator==(iterator_t q) {
13      return (bucket == q.bucket && p==q.p);
14    }
15    bool operator!=(iterator_t q) {
16      return !(*this==q);
17    }
18    iterator_t() { }
19  };
20  typedef int (*hash_fun)(key_t x);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
21
22  class hash_set {
23  private:
24    typedef std::list<key_t> list_t;
25    typedef list_t::iterator listit_t;
26    typedef std::pair<iterator_t,bool> pair_t;
27    hash_set(const hash_set&) {}
28    hash_set& operator=(const hash_set&) {}
29    hash_fun h;
30    int B;
31    int count;
32    std::vector<list_t> v;
33    iterator_t next_aux(iterator_t p) {
34      while (p.p==v[p.bucket].end()
35             && p.bucket<B-1) {
36        p.bucket++;
37        p.p = v[p.bucket].begin();
38      }
39      return p;
40    }
41  public:
42    hash_set(int B_a,hash_fun h_a)
43      : B(B_a), v(B), h(h_a), count(0) { }
44    iterator_t begin() {
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
45      iterator_t p = iterator_t(0,v[0].begin());
46      return next_aux(p);
47    }
48    iterator_t end() {
49      return iterator_t(B-1,v[B-1].end());
50    }
51    iterator_t next(iterator_t p) {
52      p.p++; return next_aux(p);
53    }
54    key_t retrieve(iterator_t p) { return *p.p; }
55    pair_t insert(const key_t& x) {
56      int b = h(x) % B;
57      list_t &L = v[b];
58      listit_t p = L.begin();
59      while (p!= L.end() && *p!=x) p++;
60      if (p!= L.end())
61        return pair_t(iterator_t(b,p),false);
62      else {
63        count++;
64        p = L.insert(p,x);
65        return pair_t(iterator_t(b,p),true);
66      }
67    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
68      iterator_t find(key_t& x) {
69        int b = h(x) % B;
70        list_t &L = v[b];
71        listit_t p = L.begin();
72        while (p!= L.end() && *p!=x) p++;
73        if (p!= L.end())
74          return iterator_t(b,p);
75        else return end();
76      }
77      int erase(const key_t& x) {
78        list_t &L = v[h(x) % B];
79        listit_t p = L.begin();
80        while (p!= L.end() && *p!=x) p++;
81        if (p!= L.end()) {
82          L.erase(p);
83          count--;
84          return 1;
85        } else return 0;
86      }
87      void erase(iterator_t p) {
88        v[p.bucket].erase(p.p);
89      }
90      void clear() {
91        count=0;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
92      for (int j=0; j<B; j++) v[j].clear();
93    }
94    int size() { return count; }
95  };
```

- Usamos `vector<>` y `list<>` de STL para implementar los vectores y listas, respectivamente.
- Los elementos a insertar en el diccionario son de tipo `key_t`.
- El `typedef hash_fun` define un tipo para las funciones admisibles como funciones de dispersión. Estas son funciones que deben tomar como argumento un elemento de tipo `key_t` y devuelve un entero.
- La clase contiene un puntero `h` a la función de dispersión. Este puntero se define en el constructor.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- El constructor toma como argumentos el número de cubetas y el puntero a la función de dispersión y los copia en los valores internos. Redimensiona el vector de cubetas `v` e inicializa el contador de elementos `count`.
- La clase iterator consiste de el número de cubeta (`bucket`) y un iterator en la lista (`p`). Las posiciones válidas son, como siempre, posiciones dereferenciables y `end()`.
- Los iterators dereferenciable consisten en un número de cubeta no vacía y una posición dereferenciable en la lista correspondiente.
- El iterator `end()` consiste en el par `bucket=B-1` y `p` el `end()` de esa lista (es decir, `v[bucket].end()`)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- **Hay que tener cuidado de, al avanzar un iterator siempre llegar a otro iterator válido. La función privada `next_aux()` avanza cualquier combinación de cubeta y posición en la lista (puede no ser válida, por ejemplo el `end()` de una lista que no es la última) hasta la siguiente posición válida.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- El tiempo de ejecución de `next_aux()` es 1 si simplemente avanza una posición en la misma cubeta sin llegar a `end()`. Si esto último ocurre entonces entra en un lazo sobre las cubetas del cual sólo sale cuando llega a una cubeta no vacía. Si el número de elementos es mayor que B entonces en promedio todas las cubetas tienen al menos un elemento y `next_aux()` a lo sumo debe avanzar a la siguiente cubeta. Es decir, el cuerpo del lazo dentro de `next_aux()` se ejecuta una sola vez. Si $n \ll B$ entonces el número de cubetas llenas es aproximadamente n y el de vacías $\approx B$ (en realidad esto no es exactamente así ya que hay una cierta probabilidad de que dos elementos vayan a la misma cubeta). Entonces, por cada cubeta llena hay B/n cubetas vacías y el lazo en `next_aux()` debe ejecutarse B/n veces. Finalmente, B/n da infinito para $n = 0$ y en realidad el número de veces que se ejecuta el lazo no es infinito sino B.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- `begin()` devuelve la primera posición válida en el diccionario. Para ello toma el iterator correspondiente a la posición `begin()` de la primera lista y le aplica `next_aux()`. Esto puede resultar en una posición dereferenciable o en `end()` (si el diccionario está vacío).
- `insert(x)` y `erase(x)` proceden de acuerdo a lo explicado en la sección previa.
- `next(p)` incrementa la posición en la lista. Pero con esto no basta ya que puede estar en el `end()` de una lista que no es la última cubeta. Por eso se aplica `next_aux()` que sí avanza a la siguiente posición válida.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Diccionarios. Tiempos de ejecución

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tiempos de ejecución

Método	$T(n, B)$ **(promedio)**	$T(n, B)$ **(peor caso)**
`retrieve(p)`, `erase(p)`, `end()`	$O(1)$	$O(1)$
`insert(x)`, `erase(x)`	$O(1 + n/B)$	$O(n)$
`begin()`, `next(p)`	$\begin{cases} \textbf{si } n = 0, & O(B); \\ \textbf{si } n <= B, & O(B/n); \\ \textbf{si } n > B & O(1); \end{cases}$	$O(B)$
`clear()`	$O(n + B)$	$O(n + B)$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones de dispersión

De los costos que hemos analizado para las tablas de dispersión abierta se deduce que para que éstas sean efectivas los elementos deben ser distribuidos uniformemente sobre las cubetas. El diseño de una buena función de dispersión es precisamente ése. Pensemos por ejemplo en el caso de una tabla de dispersión para strings con B=256 cubetas. Como función de dispersión podemos tomar

$$h_1(x) = \text{(Código ASCII del primer caracter de } x\text{)}$$

Si ésta función de dispersión es usada con strings que provienen por ejemplo de palabras encontradas encontradas en texto usual en algún lenguaje como español, entonces es probable que haya muchas más palabras que comiencen con la letra `a` y por lo tanto vayan a la cubeta 97 (el valor ASCII de `a`) que con la letra `x` (cubeta 120).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Funciones de dispersión (cont.)

```
1 int h2(string s) {
2   int v = 0;
3   for (int j=0; j<s.size(); j++) {
4     v += s[j];
5     v = v % 256;
6   }
7   return v;
8 }
```

Esta función calcula la suma de los códigos ASCII de todos los caracteres del string, módulo 256. Notamos primero que basta con que dos strings tengan un sólo caracter diferente para que sus valores de función de dispersión sean diferentes. Por ejemplo, los strings `argonauta` y `argonautas` irán a diferentes cubetas ya que difieren en un caracter. Sin embargo las palabras `vibora` y `bravio` irán a la misma ya que los caracteres son los mismos, pero en diferente orden (son anagramas la una de la otra).

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión cerradas

Otra posibilidad para resolver el problema de las colisiones es usar otra cubeta cercana a la que indica la función de dispersión. A estas estrategias se les llama de *"redispersión"*. La tabla es un `vector<key_t>` e inicialmente todos los elementos están inicializados a un valor especial que llamaremos `undef` (*"indefinido"*). Si el diccionario guarda valores enteros positivos podemos usar 0 como `undef`. Si los valores son reales (`double`'s o `float`'s) entonces podemos usar como `undef` los valores `DBL_MAX` o `NAN` (definidos en el header `float.h`). (`DBL_MAX` es el valor más grande representable en esa computadora; en el caso de un procesador de 32bits con el SO GNU/Linux y el compilador GCC es `1.7976931348623157e+308`).

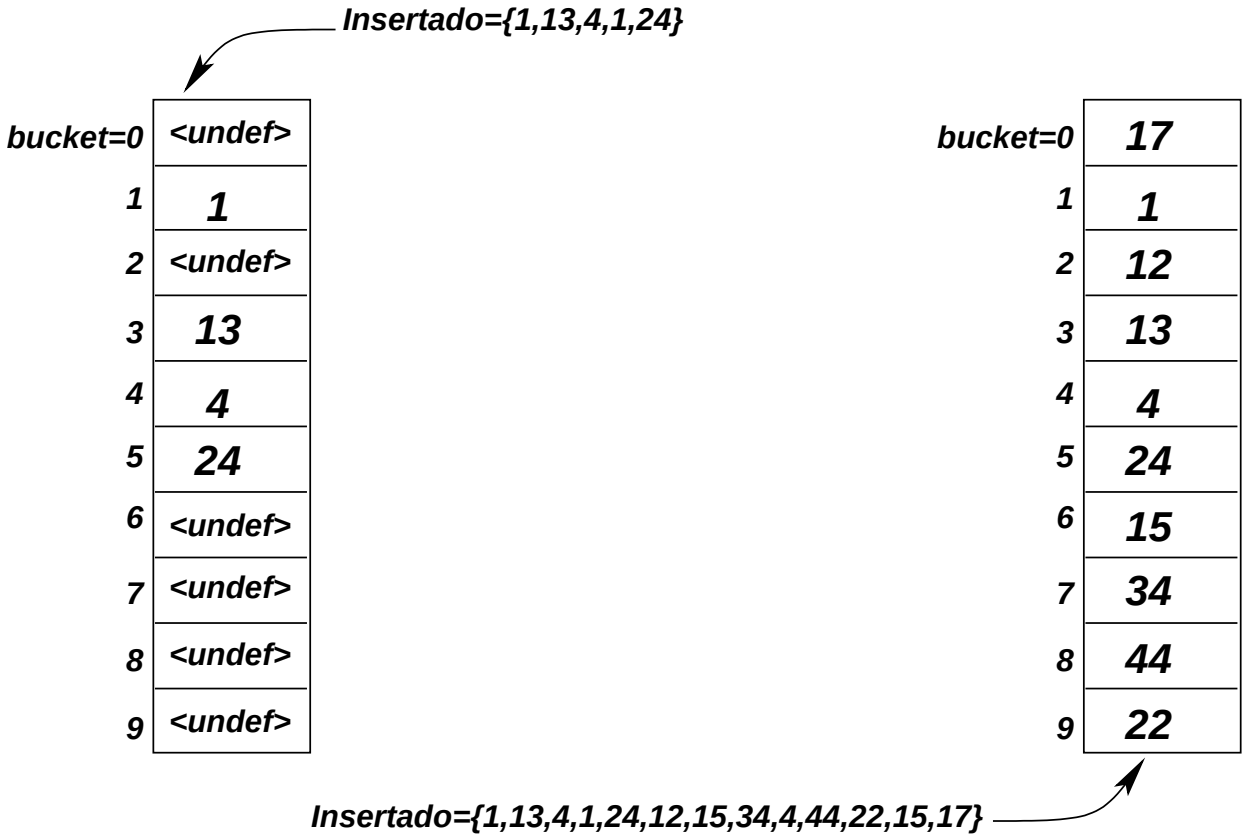
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión cerradas (cont.)

Si estamos almacenando nombres de personas podemos usar la cadena `<NONE>` (esperando que nadie se llame de esa forma) y así siguiendo. Para insertar un elemento x calculamos la función de dispersión `init`=$h(x)$ para ver cuál cubeta le corresponde *inicialmente*. Si la cubeta está libre (es decir el valor almacenado es `undef`), entonces podemos insertar el elemento en esa posición. Si está ocupado, entonces podemos probar en la siguiente cubeta (en *"sentido circular"*, es decir si la cubeta es $B-1$ la siguientes es la 0) `(init+1)%B`. Si está libre lo insertamos allí, si está ocupada y el elemento almacenado no es x, seguimos con la siguiente, etc... hasta llegar a una libre. Si todas las cubetas están ocupadas entonces la tabla está llena y el programa debe señalar un error (o agrandar la tabla dinámicamente).

Tablas de dispersión cerradas (cont.)

Ej: $B = 10$, dispersión lineal ($h(x) = x$), insertar 1,13,4,1,24,12,15,34,4,44,22,15,17,19.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Tablas de dispersión cerradas (cont.)

Ahora consideremos que ocurre al hacer `p=find(x)`. Por supuesto no basta con buscar en la cubeta $h(x)$ ya que si al momento de insertar esa cubeta estaba llena, entonces el algoritmo de inserción lo insertó en la siguiente cubeta vacía, es decir que x puede estar en otra cubeta. Sin embargo, no hace falta revisar todas las cubetas, si revisamos las cubetas a partir de $h(x)$, es decir la $h(x)+1$, $h(x)+2$... entonces cuando lleguemos a alguna cubeta que contiene a x podemos devolver el iterator correspondiente. Pero también si llegamos a un `undef`, sabemos que el elemento *"no está"* en el diccionario, ya que si estuviera debería haber sido insertado en alguna cubeta entre $h(x)$ y el siguiente `undef`. De manera que al encontrar el primer `undef` podemos retornar `end()`.

Supresiones? más adelante...

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de la inserción exitosa

Definimos la *"tasa de ocupación"* α como

$$\alpha = \frac{n}{B}$$

El costo de insertar un nuevo elemento (una *"inserción exitosa"*) es proporcional al número m de cubetas ocupadas qua hay que recorrer hasta encontrar una cubeta libre. Cuando $\alpha \ll 1$, (muy pocas cubetas ocupadas) la probabilidad de encontrar una cubeta libre es grande y el costo de inserción es $O(1)$. A medida que la tabla se va llenando la probabilidad de encontrar una serie de cubetas ocupadas es alta y el costo de inserción ya no es $O(1)$.

Costo de la inserción exitosa (cont.)

Tabla=$\{$undef, 1, undef, 13, 4, 24, undef, undef, undef, undef$\}$

Consideremos la cantidad de cubetas ocupadas que debemos recorrer hasta encontrar una cubeta libre. Consideremos que la probabilidad de que una dada cubeta sea la cubeta inicial es la misma para todas las cubetas. Si la cubeta inicial es una vacía (como las 0,2,6,7,8,9) entonces no hay que recorrer ninguna cubeta ocupada, es decir $m = 0$. Si queremos insertar en la cubeta 1, entonces ésta está ocupada, pero la siguiente está vacía, con lo cual debemos recorrer $m = 1$ cubetas. El peor caso es al querer insertar en una cubeta como la 3, para la cual hay que recorrer $m = 3$ cubetas antes de encontrar la siguiente vacía, que es la 6. Para las cubetas 4 y 5 tenemos $m = 2$ y 1 respectivamente. El m promedio (que denotaremos por $\langle m \rangle$ es la suma de los m para cada cubeta dividido el número de cubetas. En este caso es $(1 + 3 + 2 + 1)/10 = 0.7$.

Costo de la inserción exitosa (cont.)

Si todas las cubetas tienen la misma probabilidad de estar ocupadas α, entonces la probabilidad de que al insertar un elemento ésta este libre ($m = 0$ intentos) es $P(0) = 1 - \alpha$. Para que tengamos que hacer un sólo intento debe ocurrir que la primera esté llena y la siguiente vacía. La probabilidad de que esto ocurra es $P(1) = \alpha(1 - \alpha)$. Para dos intentos, la probabilidad es $\alpha^2(1 - \alpha)$ y así siguiendo, la probabilidad de que haya que hacer m intentos es

$$P(m) = \alpha^m(1 - \alpha)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Nro. de intentos infructuosos (m)	**Probabilidad de ocurrencia** $P(m) = \alpha^m(1-\alpha), \ (\alpha = 0.75)$
0	**0.250000**
1	**0.187500**
2	**0.140625**
3	**0.105469**
4	**0.079102**
5	**0.059326**
6	**0.044495**
7	**0.033371**
8	**0.025028**

Costo de la inserción exitosa (cont.)

La cantidad de intentos en promedio será entonces

$$\langle m \rangle = \sum_{m=0}^{B-1} m\, P(m)$$

$$= \sum_{m=0}^{B-1} m\, \alpha^m (1-\alpha)$$

Suponiendo que B es relativamente grande y α no está demasiado cerca de uno, podemos reemplazar la suma por una suma hasta $m = \infty$, ya que los términos decrecen muy fuertemente al crecer m.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de la inserción exitosa (cont.)

Para hacer la suma necesitamos hacer un truco matemático, para llevar la serie de la forma $m\alpha^m$ a una geométrica simple, de la forma α^m

$$m\alpha^m = \alpha \frac{\mathrm{d}}{\mathrm{d}\alpha} \alpha^m$$

de manera que

$$\begin{aligned}
\langle m \rangle &= \sum_{m=0}^{\infty} (1-\alpha)\, \alpha \frac{\mathrm{d}}{\mathrm{d}\alpha} \alpha^m \\
&= \alpha\,(1-\alpha) \frac{\mathrm{d}}{\mathrm{d}\alpha} \left(\sum_{k=0}^{\infty} \alpha^m \right) \\
&= \alpha\,(1-\alpha) \frac{\mathrm{d}}{\mathrm{d}\alpha} \left(\frac{1}{1-\alpha} \right) = \frac{\alpha}{1-\alpha}
\end{aligned}$$

Por ejemplo, en el caso de tener $B = 100$ cubetas y $\alpha = 0.9$ (90% de cubetas ocupadas) el número de intentos medio es de $\langle m \rangle = 0.9/(1-0.9) = 9$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de la inserción no exitosa

Llamaremos una *"inserción no exitosa"* cuando al insertar un elemento este ya está en el diccionario y por lo tanto en realidad no hay que insertarlo. El costo en este caso es menor ya que no necesariamente hay que llegar hasta una cubeta vacía, el elemento puede estar en cualquiera de las cubetas ocupadas. El número de intentos también depende de en qué momento fue insertado el elemento. Si fue insertado en los primeros momentos, cuando la tabla estaba vacía (α pequeños), entonces es menos probable que el elemento esté en el segmento final de largas secuencias de cubetas llenas.

Tabla=$\{$undef, 1, undef, 13, 4, 24, undef, undef, undef, undef$\}$

Ej: el elemento 4 que fue insertado al principio necesita $m = 0$ intentos infructuosos (es encontrado de inmediato), mientras que el 24, que fue insertado después, necesitará $m = 1$.

Costo de la inserción no exitosa (cont.)

Si la tabla tiene una tasa de ocupación α, entonces un elemento x puede haber sido insertado en cualquier momento previo en el cual la tasa era α', con $0 \leq \alpha' \leq \alpha$. Si fue insertado al principio ($\alpha' \approx 0$) entonces habrá que hacer pocos intentos infructuosos para encontrarlo, si fue insertado recientemente ($\alpha' \approx \alpha$) entonces habrá que hacer el mismo número promedio de intentos infructuosos que el que hay que hacer ahora para insertar un elemento nuevo, es decir $\alpha/(1-\alpha)$. Si asumimos que el elemento pudo haber sido insertado en cualquier momento cuando $0 \leq \alpha' \leq \alpha$, entonces el número de intentos infructuosos promedio será

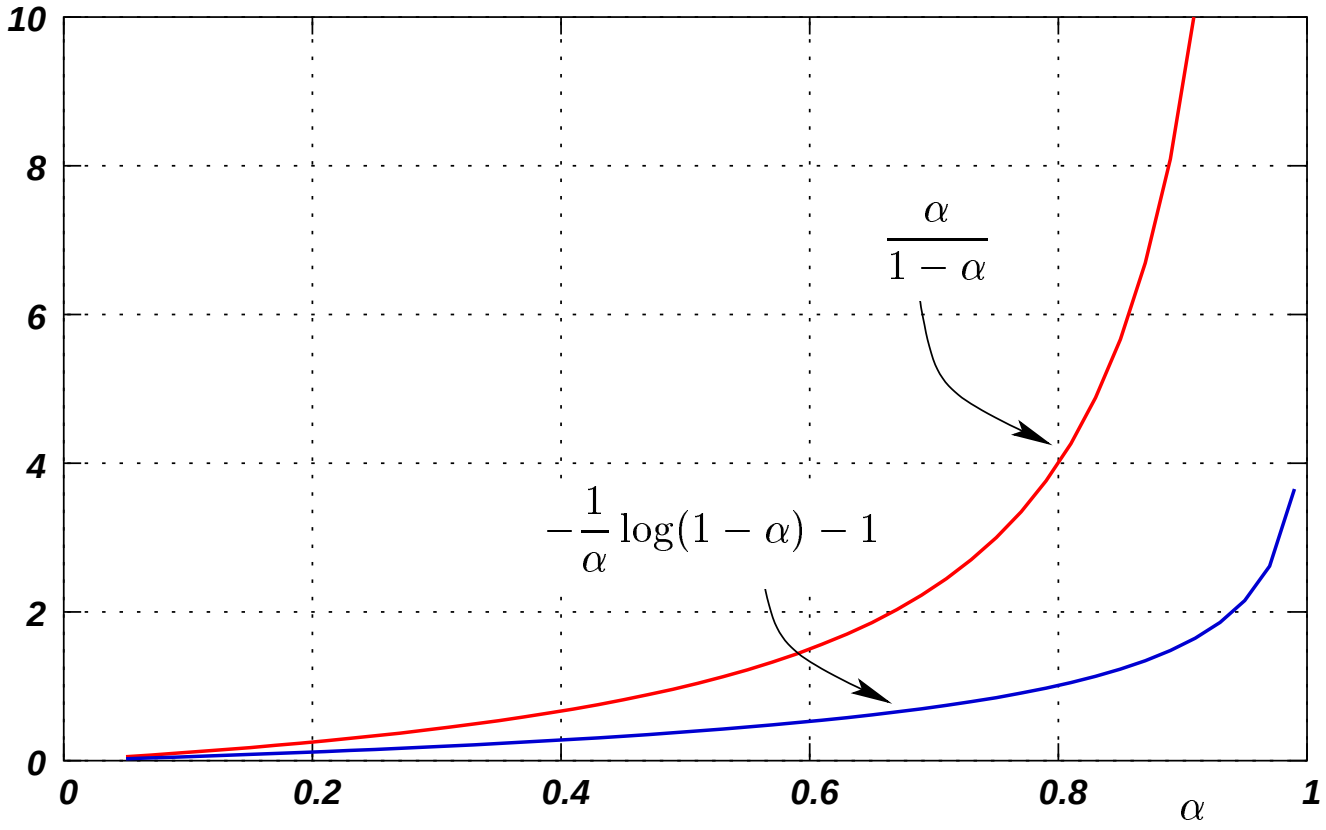
$$\langle m_{\text{n.e.}} \rangle = \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \langle m \rangle \, \mathrm{d}\alpha'$$

Reemplazando $\langle m \rangle$, tenemos que

$$\begin{aligned}
\langle m_{\text{n.e.}} \rangle &= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \frac{\alpha'}{1-\alpha'} \, \mathrm{d}\alpha' \\
&= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \left(\frac{1}{1-\alpha'} - 1 \right) \, \mathrm{d}\alpha' \\
&= \frac{1}{\alpha} \left(-\log(1-\alpha') - \alpha' \right) \Big|_{\alpha'=0}^{\alpha} \\
&= -\frac{1}{\alpha} \log(1-\alpha) - 1
\end{aligned}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de la inserción no exitosa (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de la búsqueda

Ahora analicemos el costo de `p=find(k)`. El costo es $O(1 + \langle m \rangle)$, donde $\langle m \rangle$ es el número de intentos infructuosos. Si el elemento no está en el diccionario (*búsqueda no exitosa*), entonces el número de intentos infructuosos es igual al de intentos infructuosos para inserción exitosa. Por otra parte, si el elemento está en el diccionario (*búsqueda exitosa*), el número de intentos infructuosos es sensiblemente menor, ya que si el elemento fue insertado al comienzo, cuando la tabla estaba vacía es probable que esté en las primeras cubetas, bien cerca de $h(x)$. El análisis es similar al de la inserción no exitosa y resulta ser

$$\langle m_{\text{busq.n.e.}} \rangle = \frac{\alpha}{1-\alpha}, \quad \langle m_{\text{busq.e.}} \rangle = -\frac{1}{\alpha}\log(1-\alpha) - 1$$

Costo de la búsqueda (cont.)

Supresión de elementos

Al eliminar un elemento uno estaría tentado de reemplazar el elemento por un `undef`. Sin embargo, esto dificultaría las posibles búsquedas futuras ya que ya no sería posible detenerse al encontrar un `undef` para determinar que el elemento no está en la tabla. La solución es introducir otro elemento `deleted` (*"eliminado"*) que marcará posiciones donde previamente hubo alguna vez un elemento que fue eliminado. Por ejemplo, para enteros positivos podríamos usar `undef`=0 y `deleted`=-1. Ahora, al hacer `p=find(x)` debemos recorrer las cubetas siguientes a $h(x)$ hasta encontrar x o un elemento `undef`. Los elementos `deleted` son tratados como una cubeta ocupada más. Sin embargo, al hacer un `insert(x)` de un nuevo elemento, podemos insertarlo en posiciones `deleted` además de `undef`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Costo de las funciones cuando hay supresión

Ahora la tasa de ocupación debe incluir a los elementos `deleted`, es decir en base a la *"tasa de ocupación efectiva"* α' dada por

$$\alpha' = \frac{n + n_{\text{del}}}{B}$$

donde ahora n_{del} es el número de elementos `deleted` en la tabla.

Supongamos una tabla con $B =$100 cubetas en la cual se insertan 50 elementos distintos, y a partir de allí se ejecuta un lazo infinito en el cual se inserta un nuevo elemento al azar y se elimina otro del conjunto, también al azar. Después de cada ejecución del cuerpo del lazo el número de elementos se mantiene en $n = 50$ ya que se inserta y elimina un elemento. La tabla nunca se llena, pero el número de suprimidos n_{del} crece hasta que eventualmente llega a ser igual $B - n$, es decir todas las cubetas están ocupadas o bien tienen `deleted`. Cada operación recorre toda la tabla, ya que en ningún momento encuentra un `undef`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reinserción de la tabla

Si el destino de la tabla es para insertar una cierta cantidad de elementos y luego realizar muchas consultas, pero pocas inserciones/supresiones, entonces el esquema presentado hasta aquí es razonable. Por el contrario, si el ciclo de inserción supresión sobre la tabla va a ser continuo, el incremento en el número de elementos `deleted` causará tal deterioro en la eficiencia que no permitirá un uso práctico de la tabla. Una forma de corregir esto es *"reinsertar"* los elementos, es decir, se extraen todos los elementos guardándolos en un contenedor auxiliar (vector, lista, etc...) limpiando la tabla, y reinsertando todos los elementos del contenedor.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Reinserción de la tabla (cont.)

Como inicialmente la nueva tabla tendrá todos los elementos `undef`, y las inserciones no generan elementos `deleted`, la tabla reinsertada estará libre de `deleted`'s. Esta tarea es $O(B+n)$ y se ve compensada por el tiempo que se ahorrará en unas pocas operaciones.

$$\beta = \frac{n_{\mathrm{del}}}{B}$$

Cuando $\beta \approx 1-\alpha$ quiere decir que de la fracción de cubetas no ocupadas $1-\alpha$, una gran cantidad de ellas dada por la fracción β está ocupada por suprimidos, degradando la eficiencia de la tabla. Por ejemplo, si $\alpha = 0.5$ y $\beta = 0.45$ entonces 50% de las cubetas está ocupada, y del restante 50% el 45% está ocupado por `deleted`. En esta situación la eficiencia de la tabla es equivalente una con un 95% ocupado.

Reinserción continua

	S={}	S={24}	S={24}	S={24}
bucket=0	<undef>	0 <undef>	0 <undef>	0 <undef>
1	1	1 1	1 1	1 1
2	<undef>	2 <undef>	2 <undef>	2 <undef>
3	13	3 13	3 13	3 13
4	4	4 4	4 <undef>	4 24
5	24	5 <undef>	5 <undef>	5 <undef>
6	<undef>	6 <undef>	6 <undef>	6 <undef>
7	<undef>	7 <undef>	7 <undef>	7 <undef>
8	<undef>	8 <undef>	8 <undef>	8 <undef>
9	<undef>	9 <undef>	9 <undef>	9 <undef>

En la reinserción continua, cada vez que hacemos un borrado hacemos una series de operaciones para dejar la tabla sin ningún `deleted`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estrategias de redispersión

Al introducir las tablas de dispersión cerrada hemos explicado como la redispersión permite resolver las colisiones, buscando en las cubetas $h(x)+j$, para $j=0,1,...,B-1$, hasta encontrar una cubeta libre. A esta estrategia se le llama *"de redispersión lineal"*. Podemos pensar que si la función de dispersión no es del todo buena, entonces ciertas cubetas o ciertas secuencias de cubetas cercanas pueden tender a llenarse más que otras. Esto puede ocasionar que tiendan a formarse localmente secuencias de cubetas ocupadas, incluso cuando la tasa de ocupación no es tan elevada globalmente.

Estrategias de redispersión (cont.)

En estos casos puede ayudar el tratar de no dispersar las cubetas en forma lineal sino de la forma $h(x) + d_j$, donde d_0 es 0 y $d_j, j = 1, ..., B - 1$ es una permutación de los números de 1 a $B - 1$. Por ejemplo, si $B = 8$, podríamos tomar alguna permutación aleatoria como $d_j = 7, 2, 5, 4, 1, 0, 3, 6$. La forma de generar estas permutaciones es similar a la generación de números aleatorios, sin embargo está claro que no son números aleatorios, ya que la secuencia d_j debe ser la misma durante todo el uso de la tabla.

Detalles de implementación

```
1   typedef int iterator_t;
2   typedef int (*hash_fun)(key_t x);
3   typedef int (*redisp_fun)(int j);
4
5   int linear_redisp_fun(int j) { return j; }
6
7   class hash_set {
8   private:
9     hash_set(const hash_set&) {}
10    hash_set& operator=(const hash_set&) {}
11    int undef, deleted;
12    hash_fun h;
13    redisp_fun rdf;
14    int B;
15    int count;
16    std::vector<key_t> v;
17    std::stack<key_t> S;
18    iterator_t locate(key_t x,iterator_t &fdel) {
19      int init = h(x)+rdf(0);
20      int bucket;
21      bool not_found = true;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22        for (int i=0; i<B; i++) {
23          bucket = (init+rdf(i)) % B;
24          key_t vb = v[bucket];
25          if (vb==x || vb==undef) break;
26          if (not_found && vb==deleted) {
27            fdel=bucket;
28            not_found = false;
29          }
30        }
31        if (not_found) fdel = end();
32        return bucket;
33      }
34      iterator_t next_aux(iterator_t bucket) {
35        int j=bucket;
36        while(j!=B && v[j]!=undef && v[j]!=deleted) {
37          j++;
38        }
39        return j;
40      }
41    public:
42      hash_set(int B_a,hash_fun h_a,
43               key_t undef_a,key_t deleted_a,
44               redisp_fun rdf_a=&linear_redisp_fun)
45        : B(B_a), undef(undef_a), v(B,undef_a), h(h_a),
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
46        deleted(deleted_a), rdf(rdf_a), count(0)
47      { }
48      std::pair<iterator_t, bool>
49        insert(key_t x) {
50        iterator_t fdel;
51        int bucket = locate(x,fdel);
52        if (v[bucket]==x)
53          return std::pair<iterator_t,bool>(bucket,false);
54        if (fdel!=end()) bucket = fdel;
55        if (v[bucket]==undef || v[bucket]==deleted) {
56          v[bucket]=x;
57          count++;
58          return std::pair<iterator_t,bool>(bucket,true);
59        } else {
60          std::cout << "Tabla de dispersion llena!!\n";
61          abort();
62        }
63      }
64      key_t retrieve(iterator_t p) { return v[p]; }
65      iterator_t find(key_t x) {
66        iterator_t fdel;
67        int bucket = locate(x,fdel);
68        if (v[bucket]==x) return bucket;
69        else return(end());
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
70      }
71      int erase(const key_t& x) {
72        iterator_t fdel;
73        int bucket = locate(x,fdel);
74        if (v[bucket]==x) {
75          v[bucket]=deleted;
76          count--;
77            // Trata de purgar elementos 'deleted'
78            // Busca el siguiente elemento 'undef'
79            int j;
80            for (j=1; j<B; j++) {
81              op_count++;
82              int b = (bucket+j) % B;
83              key_t vb = v[b];
84              if (vb==undef) break;
85              S.push(vb);
86              v[b]=undef;
87              count--;
88            }
89            v[bucket]=undef;
90            // Va haciendo erase/insert de los elementos
91            // de atras hacia adelante hasta que se llene
92            // 'bucket'
93            while (!S.empty()) {
94              op_count++;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
95            insert(S.top());
96            S.pop();
97          }
98        return 1;
99      } else return 0;
100   }
101   iterator_t begin() {
102     return next_aux(0);
103   }
104   iterator_t end() { return B; }
105   iterator_t next(iterator_t p) {
106     return next_aux(p++);
107   }
108   void clear() {
109     count=0;
110     for (int j=0; j<B; j++) v[j]=undef;
111   }
112   int size() { return count; }
113 };
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- El `typedef hash_fun` define el tipo de funciones que pueden usarse como funciones de dispersión (toman como argumento un elemento de tipo `key_t` y devuelven un entero). La función a ser usada es pasada en el constructor y guardada en un miembro `hash_fun h`.
- El `typedef redisp_fun` define el tipo de funciones que pueden usarse para la redispersión (los d_j). Es una función que debe tomar como argumento un entero j y devolver otro entero (el d_j). La función a ser usada es pasada por el usuario en el constructor y guardada en un miembro (`redisp_fun rdf`). Por defecto se usa la función de redispersión lineal (`linear_redisp_fun()`).
- El miembro dato `int B` es el número de cubetas a ser usada. Es definido en el constructor.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- El miembro dato `int count` va a contener el número de elementos en la tabla (será retornado por `size()`).
- Las B cubetas son almacenadas en un `vector<key_t> v`.
- Casi todos los métodos de la clase usan la función auxiliar `iterator_t locate(key_t x,iterator_t &fdel)`. Esta función retorna un iterator de acuerdo a las siguiente reglas:
  - Si `x` está en la tabla `locate()` retorna la cubeta donde está almacenado el elemento `x`.
  - Si `x` no está en la tabla, retorna el primer `undef` después (en sentido circular) de la posición correspondiente a `x`.
  - Si no hay ningún `undef` (pero puede haber `deleted`) retorna alguna posición no especificada en la tabla.
  - Retorna por el argumento `fdel` la posición del primer `deleted` entre la cubeta correspondiente a `x` y el primer `undef`. Si no existe ningún `deleted` entonces retorna `end()`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- La pila `S` es usada como contenedor auxiliar para la estrategia de reinserción.
- La clase iterator consiste simplemente de un número de cubeta. `end()` es la cubeta ficticia `B`. La función `q=next(p)` que avanza iteradores, debe avanzar sólo sobre ***cubetas ocupadas***. La función `next_aux()` avanza un iterador (en principio inválido) hasta llegar a uno ocupado o a `end()`. La función `q=next(p)` simplemente incrementa `p` y luego aplica `next_aux()`.
- La función `begin()` debe retornar el primer iterator válido (la primera cubeta ocupada) o bien `end()`. Para ello calcula el `next_aux(0)` es decir la primera cubeta válida después de (o igual a) la cubeta 0.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

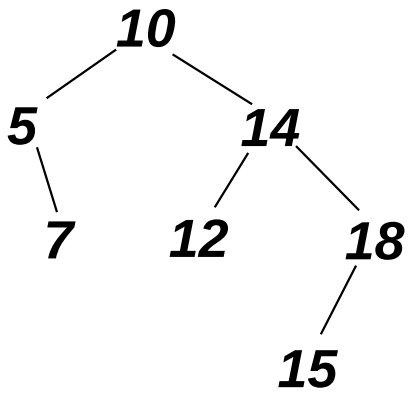
Conjuntos con árboles binarios de búsqueda

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios de búsqueda

Una forma muy eficiente de representar conjuntos son los árboles binarios de búsqueda (ABB). Un árbol binario es un ABB si es vacío (Λ) o:

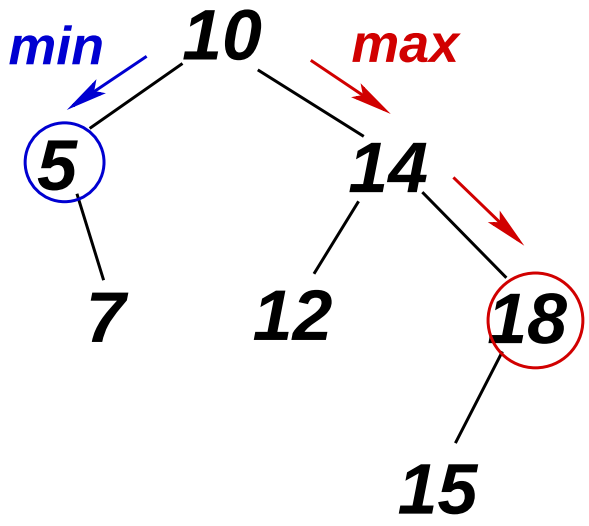
- **Todos los elementos en los nodos del subárbol izquierdo son menores que el nodo raíz.**
- **Todos los elementos en los nodos del subárbol derecho son mayores que el nodo raíz.**
- **Los subárboles del hijo derecho e izquierdo son a su vez ABB.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios de búsqueda (cont.)

- **Para buscar el máximo se debe avanzar siempre por la derecha. Para encontrar el mínimo siempre por la izquierda. Costo $O(\text{máx prof}) = O(\log_2 n)$ (si el árbol esta bien balanceado).**
- **El listado en orden simétrico da la lista ordenada de los elementos del árbol.**



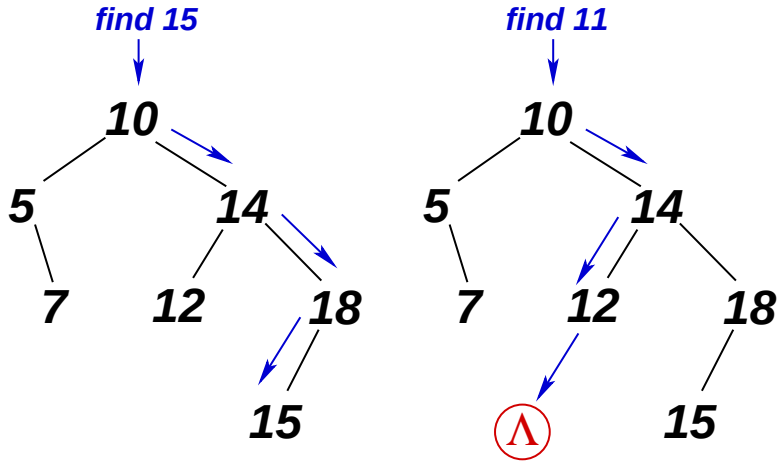
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios de búsqueda (cont.)

Buscar un elemento es recursivo

```
1  node_t find(tree_t t,node_t n,T x) {
2    if (n==t.end()) return t.end();
3    if (x<*n) return find(t,n.left(),x)
4    elsif (x>*n) return find(t,n.right(),x)
5    else return n;
6  }
```

También es $O(\text{máx prof}) = O(\log_2 n)$ ya que sigue un camino en el árbol.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

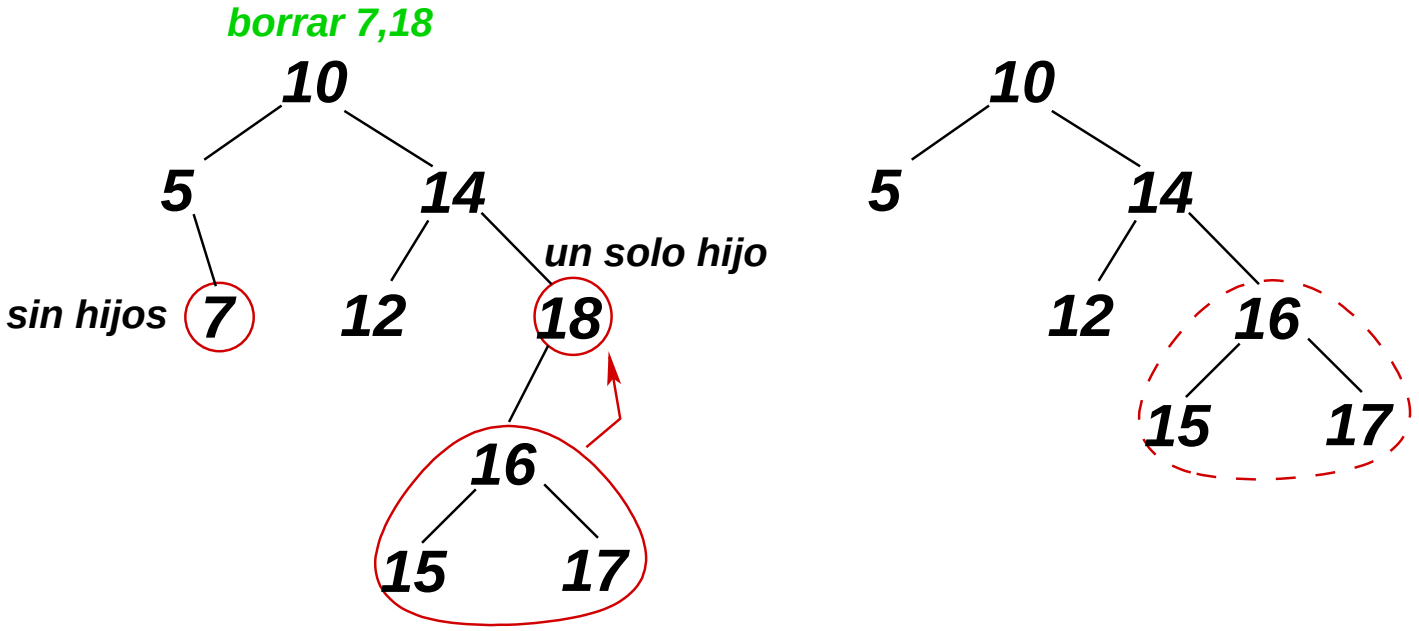
Arboles binarios de búsqueda (cont.)

Para insertar un elemento `x` hacemos un `find(x)`, si el nodo retornado es Λ insertamos allí, si no el elemento ya está. La inserción es $O(1)$ de manera que toda la operación es básicamente la del find: $O(\text{máx prof}) = O(\log_2 n)$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

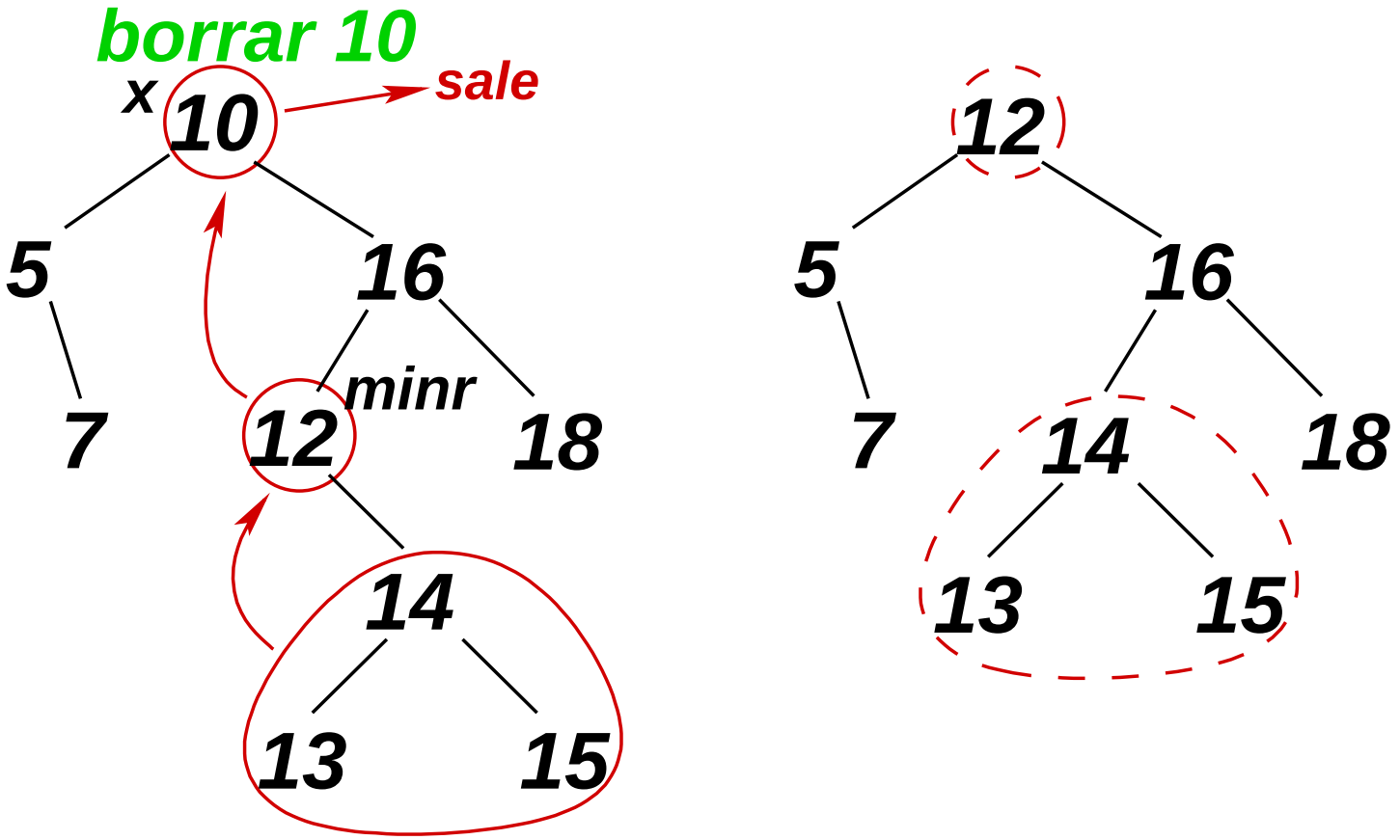
Arboles binarios de búsqueda (cont.)

- Para borrar hacemos de nuevo el `n=find(x)`. Si el elemento no está, no hay que hacer nada. ($O(\text{máx prof}) = O(\log_2 n)$).
- Si está hay varios casos, dependiendo del número de hijos.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

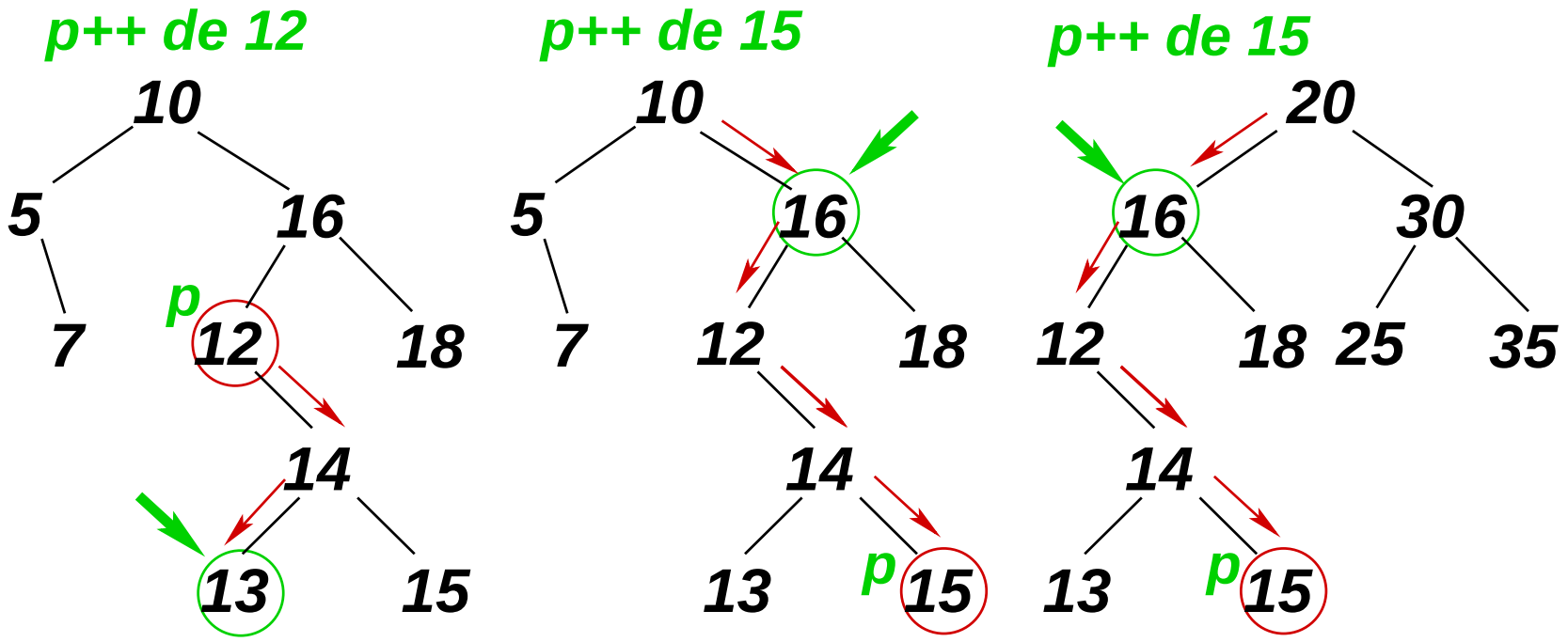
Arboles binarios de búsqueda (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Arboles binarios de búsqueda (cont.)

`next()`: (`operator++`) min del hijo derecho o primer antecesor a la derecha.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación

```
1   template<class T>
2   class set {
3   private:
4     typedef btree<T> tree_t;
5     typedef typename tree_t::iterator node_t;
6     tree_t bstree;
7     node_t min(node_t m) {
8       if (m == bstree.end()) return bstree.end();
9       while (true) {
10        node_t n = m.left();
11        if (n==bstree.end()) return m;
12        m = n;
13      }
14    }
15
16    void set_union_aux(tree_t &t,node_t n) {
17      if (n==t.end()) return;
18      else {
19        insert(*n);
20        set_union_aux(t,n.left());
21        set_union_aux(t,n.right());
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22        }
23      }
24      void set_intersection_aux(tree_t &t,
25                                node_t n, set &B) {
26        if (n==t.end()) return;
27        else {
28          if (B.find(*n)!=B.end()) insert(*n);
29          set_intersection_aux(t,n.left(),B);
30          set_intersection_aux(t,n.right(),B);
31        }
32      }
33      void set_difference_aux(tree_t &t,
34                              node_t n, set &B) {
35        if (n==t.end()) return;
36        else {
37          if (B.find(*n)==B.end()) insert(*n);
38          set_difference_aux(t,n.left(),B);
39          set_difference_aux(t,n.right(),B);
40        }
41      }
42      int size_aux(tree_t t,node_t n) {
43        if (n==t.end()) return 0;
44        else return 1+size_aux(t,n.left())
45                    +size_aux(t,n.right());
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
46      }
47    public:
48      class iterator {
49      private:
50        friend class set;
51        node_t node;
52        tree_t *bstree;
53        iterator(node_t m,tree_t &t)
54          : node(m), bstree(&t) {}
55        node_t next(node_t n) {
56          node_t m = n.right();
57          if (m!=bstree->end()) {
58            while (true) {
59              node_t q = m.left();
60              if (q==bstree->end()) return m;
61              m = q;
62            }
63          } else {
64            // busca el padre
65            m = bstree->begin();
66            if (n==m) return bstree->end();
67            node_t r = bstree->end();
68            while (true) {
69              node_t q;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
70            if (*n<*m) { q = m.left(); r=m; }
71            else q = m.right();
72            if (q==n) break;
73            m = q;
74          }
75          return r;
76        }
77      }
78    public:
79      iterator() : bstree(NULL) { }
80      iterator(const iterator &n)
81        : node(n.node), bstree(n.bstree) {}
82      iterator& operator=(const iterator& n) {
83        bstree=n.bstree;
84        node = n.node;
85      }
86      T &operator*() { return *node; }
87      T *operator->() { return &*node; }
88      bool operator!=(iterator q) {
89        return node!=q.node; }
90      bool operator==(iterator q) {
91        return node==q.node; }
92
93      // Prefix:
94      iterator operator++() {
95        node = next(node);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
96        return *this;
97      }
98      // Postfix:
99      iterator operator++(int) {
100       node_t q = node;
101       node = next(node);
102       return iterator(q, *bstree);
103     }
104   };
105 private:
106   typedef pair<iterator,bool> pair_t;
107 public:
108   set() {}
109   set(const set &A) : bstree(A.bstree) {}
110   ~set() {}
111   pair_t insert(T x) {
112     node_t q = find(x).node;
113     if (q == bstree.end()) {
114       q = bstree.insert(q,x);
115       return pair_t(iterator(q,bstree),true);
116     } else return pair_t(iterator(q,bstree),false);
117   }
118   void erase(iterator m) {
119     node_t p = m.node;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
120       node_t qr = p.right(),
121         ql = p.left();
122       if (qr==bstree.end() && ql==bstree.end())
123         bstree.erase(p);
124       else if (qr == bstree.end()) {
125         btree<T> tmp;
126         tmp.splice(tmp.begin(),ql);
127         bstree.erase(p);
128         bstree.splice(p,tmp.begin());
129       } else if (ql == bstree.end()) {
130         btree<T> tmp;
131         tmp.splice(tmp.begin(),p.right());
132         bstree.erase(p);
133         bstree.splice(p,tmp.begin());
134       } else {
135         node_t r = min(qr);
136         T minr = *r;
137         erase(iterator(r,bstree));
138         *p = minr;
139       }
140     }
141     int erase(T x) {
142       iterator q = find(x);
143       int ret;
144       if (q==end()) ret = 0;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
145       else {
146         erase(q);
147         ret = 1;
148       }
149       return ret;
150     }
151     void clear() { bstree.clear(); }
152     iterator find(T x) {
153       node_t m = bstree.begin();
154       while (true) {
155         if (m == bstree.end())
156           return iterator(m,bstree);
157         if (x<*m) m = m.left();
158         else if (x>*m) m = m.right();
159         else return iterator(m,bstree);
160       }
161     }
162     iterator begin() {
163       return iterator(min(bstree.begin()),bstree);
164     }
165     iterator end() {
166       return iterator(bstree.end(),bstree);
167     }
168     int size() {
169       return size_aux(bstree,bstree.begin()); }
170     friend void
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
171     set_union<>(set<T> &A,set<T> &B,set<T> &C);
172     friend void
173     set_intersection<>(set<T> &A,set<T> &B,set<T> &C);
174     friend void
175     set_difference<>(set<T> &A,set<T> &B,set<T> &C);
176   };
177
178   template<class T> void
179   set_union(set<T> &A,set<T> &B,set<T> &C) {
180     C.clear();
181     C.set_union_aux(A.bstree,A.bstree.begin());
182     C.set_union_aux(B.bstree,B.bstree.begin());
183   }
184
185   template<class T> void
186   set_intersection(set<T> &A,set<T> &B,set<T> &C) {
187     C.clear();
188     C.set_intersection_aux(A.bstree,
189                            A.bstree.begin(),B);
190   }
191
192   // C = A - B
193   template<class T> void
194   set_difference(set<T> &A,set<T> &B,set<T> &C) {
195     C.clear();
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
196     C.set_difference_aux(A.bstree,
197                          A.bstree.begin(),B);
198   }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación

- El tipo `set<T>` es un template que contiene un árbol binario `btree<T> bstree`.
- Los `typedef tree_t` y `node_t` son abreviaciones (privadas) para acceder convenientemente al árbol subyacente.
- La función `min(m)` retorna un iterator al nodo con el menor elemento del subárbol del nodo `m`. El nodo se encuentra bajando siempre por el hijo izquierdo.
- La clase iterator contiene no sólo el nodo en el árbol sino también un puntero al árbol mismo. Esto es necesario ya que algunas operaciones de árboles (por ejemplo la comparación con `end()`) necesitan tener acceso al árbol donde está el nodo.
- `begin()` utiliza `min(bstree.begin())` para encontrar el nodo con el menor elemento en el árbol.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- La implementación incluye un constructor por defecto (el conjunto esta vacío), un constructor por copia que invoca al constructor por copia de árboles. El operador de asignación es sintetizado automáticamente por el compilador y funciona correctamente ya que utiliza el operador de asignación para `btree<>` el cual fue correctamente implementado (hace una *"deep copy"* del árbol).
- En este caso no mantenemos un contador de nodos, por lo cual la función `size()` calcula el número de nodo usando una función recursiva auxiliar `size_aux()`.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

- **Las operadores de incremento para iterators (tanto prefijo como postfijo) utilizan una función auxiliar `next()` que calcula la posición correspondiente al siguiente nodo en el árbol (en el sentido ordenado). Esta función se complica un poco ya que nuestra clase árbol no cuenta con una función *"padre"*. Entonces, cuando el nodo no tiene hijo derecho y es necesario buscar el último *"padre derecho"*, el camino no se puede seguir desde abajo hacia arriba sino desde arriba hacia abajo, comenzando desde la raíz. De todas formas, el costo es $O(d)$ donde d es la altura del árbol, gracias a que la estructura de ABB nos permite ubicar el nodo siguiendo un camino.**
- **La función `erase()` es implementada eficientemente en términos de `splice()` cuando el nodo tiene sus dos hijos.**

Detalles de implementación (cont.)

- **Las unión se implementa simplemente simplemente insertando todos los elementos de `A` y `B` en `C`. Como todas las operaciones indivuduales involucradas son $O(\log n)$, el costo final es $O(n \log n)$.**
- **Para intersección se insertan recorren todos los elementos de `A` y se insertan *si están en* `B`. Para diferencia se insertan *si no están en* `B`.**
- **Las funciones binarias utilizan funciones recursivas miembros privados de la clase `set_union_aux(T,n)`, `set_intersection_aux(T,n,B)` y `set_difference_aux(T,n,B)` que aplican la estrategia explicada.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

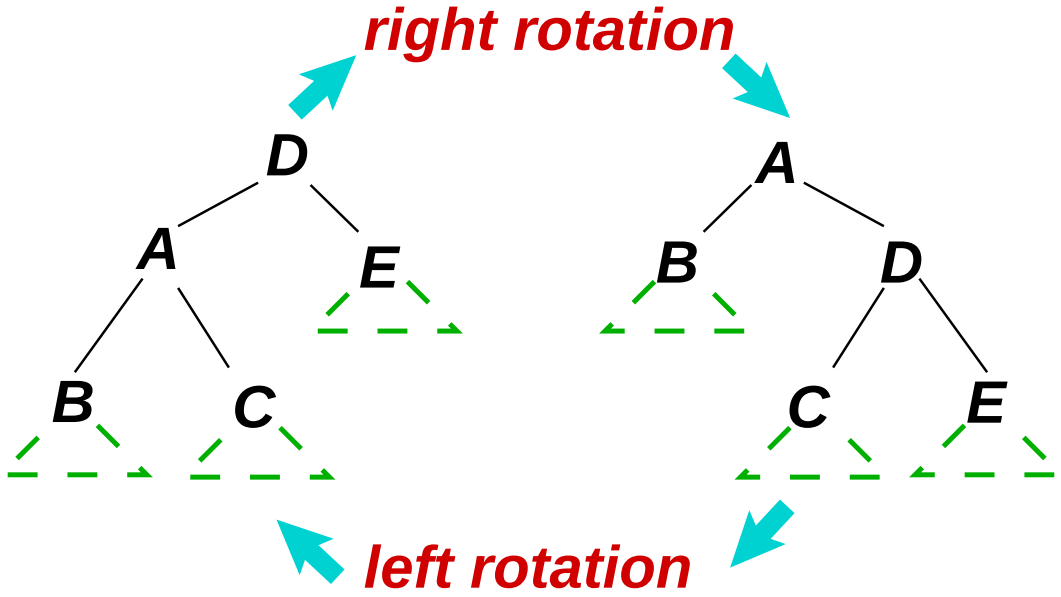
Tiempos de ejecución

Método	$T(n)$ **(promedio)**	$T(n)$ **(peor caso)**
`*p, end()`	$O(1)$	$O(1)$
`insert(x), find(x), erase(p), erase(x), begin(), p++, ++p`	$O(\log n)$	$O(n)$
`set_union(A,B,C), set_intersection(A,B,C), set_difference(A,B,C)`	$O(n \log n)$	$O(n^2)$

((version aed-2.0.4-10-gdb5d209) (date <unknown>) (processed-date Wed Dec 22 17:51:23 2010 -0300))

Balanceo del ABB

- **La eficiencia del ABB está dada directamente por el *"balanceo"* del mismo, es decir por cuanto se desvía la altura del árbol, con respecto a la media $\propto \log n$.**
- **Existen varias estrategias para mantener el ABB balanceado en forma automática.**
- ***AVL trees:* Mantienen el árbol balanceado mediante rotaciones al insertar/suprimir. Se garantiza que la altura del subárbol izquierdo de un nodo difiere a lo sumo en +1,-1 con respecto a la del derecho.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Balanceo del ABB (cont.)

- ***Treaps:*** **(Por mezcla de tree y heap). Mantienen el árbol balanceado agregando a cada nodo un elemento entero llamado prioridad. En todo momento se mantiene la propiedad de ABB y además que el árbol este parcialmente ordenado con respecto a la prioridad. Para esto se realizan rotaciones de la misma forma que para AVL. Las prioridades se genera en forma aleatoria al insertar suprimir, esto garantiza un comportamiento casi ideal. (clase `treap_set` en `treap.h`).**
- **También *red-black trees*, *splay trees*, ...**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento

El proceso de ordenar elementos (*"sorting"*) en base a alguna relación de orden es un problema tan frecuente y con tantos usos que merece un capítulo aparte. Nosotros nos concentraremos en el *ordenamiento interno*, es decir cuando todo el contenedor reside en la memoria principal de la computadora. Después daremos una introducción básica al tema del ordenamiento *externo*, donde el volumen de datos a ordenar es de tal magnitud que requiere el uso de memoria auxiliar.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento (cont.)

- **Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo `key_t` con una *relación de orden* `<` y que están almacenados en un *contenedor lineal* (vector o lista).**
- **Recordar que `vector<>` es un contenedor de *acceso aleatorio*, es decir que acceder al elemento j-ésimo es $O(1)$ mientras que para `list<>` el tiempo es $O(n)$ (en realidad $O(j)$). Pero por otra parte para listas es muy simple *dividirlas y unirlas dinámicamente*, por lo que en realidad existen técnicas especiales de ordenamiento para ambos contenedores.**

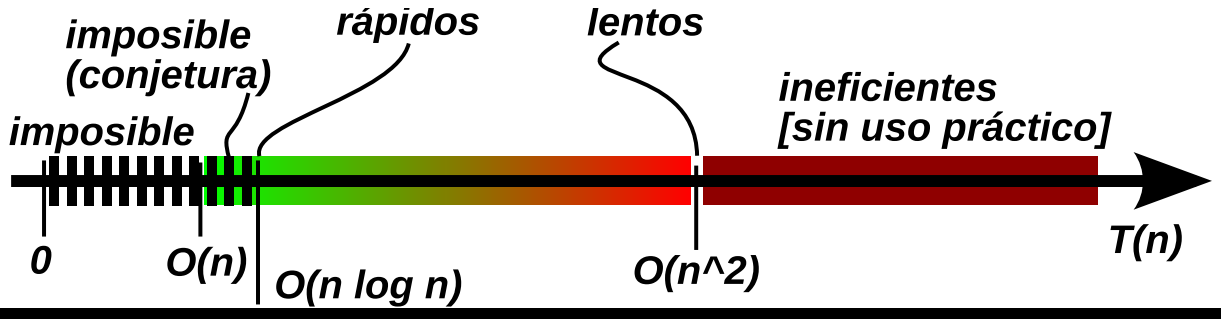
Ordenamiento (cont.)

Eficiencia de los métodos de ordenamiento

- **Por supuesto, el objetivo principal es obtener algoritmos de ordenamiento lo más *eficientes posibles*. En particular *tiempo de cálculo* y *memoria requerida*. Los tiempos de ejecución se compararán en base a su crecimiento con el tamaño del contenedor n.**
- ***Memoria adicional:* Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con n), decimos que es *"in-place"*. Si no, se debe tener en cuenta también como crece la cantidad de memoria requerida.**

Discusión básica de eficiencia

- **Recordemos la discusión previa de la complejidad algorítmica de los métodos de ordenamiento. Como piso tenemos $T(n) \geq O(n)$ ya que al menos hay que *revisar los elementos a ordenar*.**
- **Hemos visto en el capítulo 1 como ejemplo para el cálculo de los tiempos de ejecución el *Método de la Burbuja* que es $O(n^2)$. Es relativamente sencillo proponer e implementar algoritmos $O(n^2)$ y existen una gran variedad de algoritmos con estos tiempos.**
- **Existen *algoritmos más rápidos*, más adelante veremos varios con $T(n) = O(n \log n)$, e incluso en casos particulares $O(n)$. A estos los llamaremos *"rápidos"*. Hay algunos intermedios (como *"shell-sort"* que es $O(n^{1.5})$).**
- **Se *conjetura* que no hay algoritmos generales de ordenamiento con $T(n) < O(n \log n)$.**



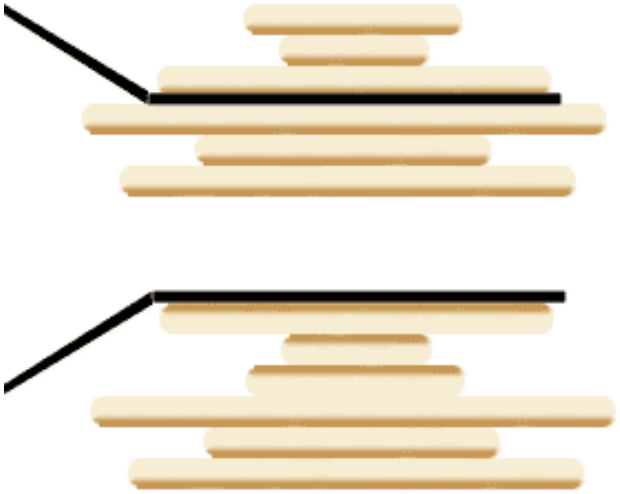
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Métodos de ordenamiento ineficientes

Algoritmos de ordenamientos *ineficientes*:

- ***Bogosort:*** **Desordenar aleatoriamente el vector hasta que quede ordenado. En promedio $T(n) = O(n \cdot n!)$, peor caso: $T(n) = \infty$.**

- ***Pancake sort:*** **Se restringe a hacer sólo operaciones en las cuales un *rango del vector prefijo* (`[0,m)` con `m<=n`) es *invertido*. (Como si fuera una pila de panqueques). Sólo tiene interés si se puede encontrar un hardware que realize las inversiones en forma muy eficiente.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad

- **Recordemos que una *relación de orden fuerte* debe ser transitiva:** $a < b$ **y** $b < c \implies a < c$ **y satisfacer que, dados** x **e** y **sólo una de las siguienes es verdadera**

$$x < y, \quad y < x, \quad x = y$$

- **Una relación de orden es *débil* si para todo** x, y **nunca ocurre que**

$$x < y, \quad \textbf{e} \quad y < x,$$

son verdaderas al mismo tiempo. Si ninguna de $x < y$ $y < x$ **son falsas entonces decimos que** x **e** y **son equivalentes (**$x \equiv y$**).**
Por ejemplo
  - ▷ **Ordenar enteros por su valor absoluto (**$-3 \equiv 3$**).**
  - ▷ **Pares de enteros por la primera componente. (**$(2, 3) \equiv (2, 5)$**).**
  - ▷ **Legajos por el nombre del empleado. (`Perez, Juan,14231235`)** $\equiv$ **(`Perez, Juan,12765987`).**

Métodos de ordenamiento lentos

- **Para ordenar asumiremos una *"relación de orden débil"*.**
- ***Estabilidad:*** **Un algoritmo de ordenamiento es estable si aquellos elementos equivalentes entre sí quedan en el orden en el que estaban originalmente. Por ejemplo, si ordenamos (-3,2,-4,5,3,-2,4) por valor absoluto, entonces podemos tener**

$$(2, -2, -3, 3, -4, 4, 5), \quad \textbf{estable}$$

$$(-2, 2, -3, 3, 4, -1, 5), \quad \textbf{no estable}$$

- **La relación de orden se pasa normalmente por una función de tipo `bool (*comp)(T,T)` donde `T` es el tipo elemento.**

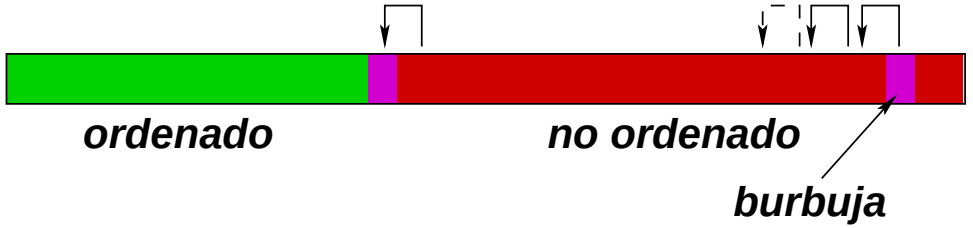
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Métodos de ordenamiento lentos

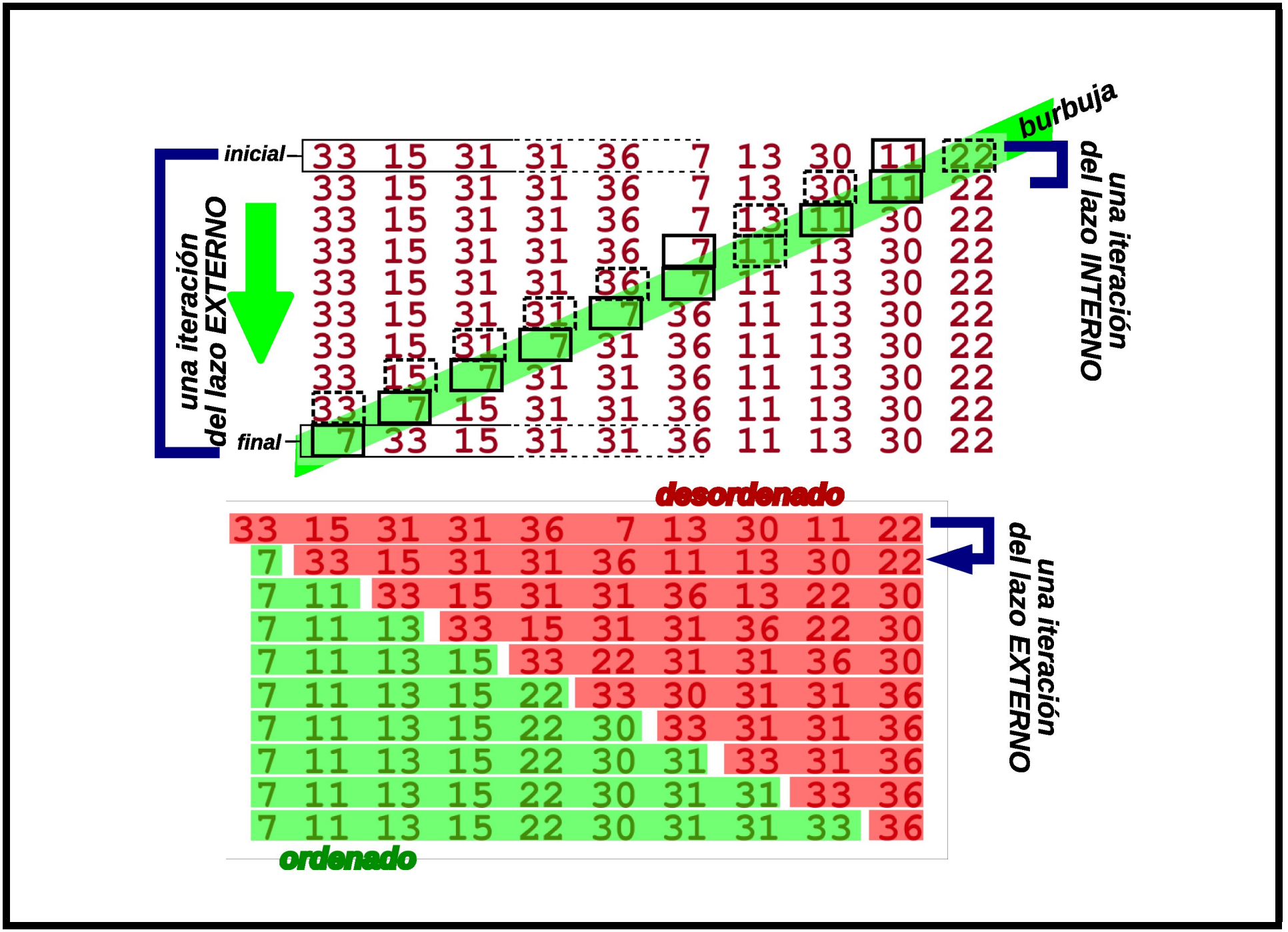
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de burbuja, "bubble-sort"

- **Hay un doble lazo. En el primer paso el elemento menor llega a la primera posición.**
- **Para eso en el lazo interno se intercambian los elementos entre la posición j y $j+1$ hasta que el elemento menor llega a la primera posición.**
- **Luego se aplica al intervalo $[1, n)$, $[2, n)$, $[3, n)$, etc...**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de burbuja, "bubble-sort" (cont.)

- Las rutinas reciben un *rango* de elementos a ordenar.
- Normalmente se hace *in-place*.
- Tenemos dos versiones de la función: *con función de comparación* y *sin función de comparación*. En el último caso usa como función de comparación la relación `<` del tipo `T`.
- En las STL sólo existe `sort()`, nosotros generaremos varias versiones, por ejemplo `bubble_sort()`, `quick_sort()`...
- Como las funciones no reciben `v` debemos usar `*(first+k)` en lugar de `v[k]`

```
1  template<class T>
2  void sort(vector<T>::iterator first,
3            vector<T>::iterator last,
4            bool (*comp)(T,T)) { /* . . . . */ }
5
6  template<class T>
7  void sort(vector<T>::iterator first,
8            vector<T>::iterator last) {
9    sort(first,last,less<T>);
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de burbuja, "bubble-sort" (cont.)

```
1   template<class T>
2   bool less(T x,T y) {
3     return x<y;
4   }
5
6   vector<int> v;
7   // Pone elementos en v. . .
8
9   sort(v.begin(),v.end()); // Ordena por < de enteros
10
11  // Ordena por valor absoluto
12  bool less_abs(int x, int y) {
13    return abs(x)<abs(y);
14  }
15  sort(v.begin(),v.end(),less_abs);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

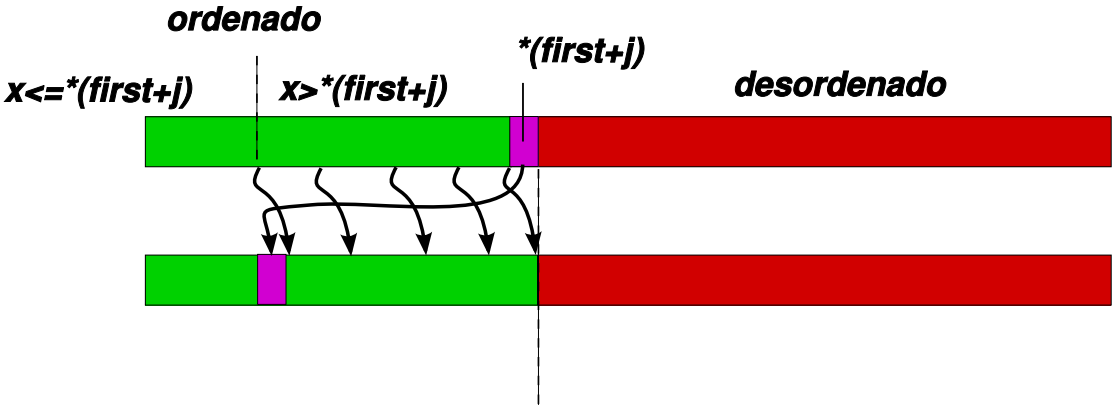
Método de burbuja, "bubble-sort" (cont.)

```
1   template<class T> void
2   bubble_sort(typename std::vector<T>::iterator first,
3               typename std::vector<T>::iterator last,
4                   bool (*comp)(T&,T&)) {
5     int size = last-first;
6     for (int j=0; j<size-1; j++) {
7       for (int k=size-1; k>j; k--) {
8         if (comp(*(first+k),*(first+k-1))) {
9           T tmp = *(first+k-1);
10          *(first+k-1) = *(first+k);
11          *(first+k) = tmp;
12        }
13      }
14    }
15  }
16
17  template<class T> void
18  bubble_sort(typename std::vector<T>::iterator first,
19              typename std::vector<T>::iterator last) {
20    bubble_sort(first,last,less<T>);
21  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de inserción, "insertion-sort"

- **Hay un doble lazo. En el lazo j el rango $[0, j)$ está ordenado e insertamos el elemento j en el rango, haciendo los desplazamientos necesarios.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de inserción, "insertion-sort" (cont.)

```
1  template<class T> void
2  insertion_sort(typename
3                 std::vector<T>::iterator first,
4                 typename
5                 std::vector<T>::iterator last,
6                 bool (*comp)(T&,T&)) {
7    int size = last-first;
8    for (int j=1; j<size; j++) {
9      T tmp = *(first+j);
10     int k=j-1;
11     while (comp(tmp,*(first+k))) {
12       *(first+k+1) = *(first+k);
13       if (--k < 0) break;
14     }
15     *(first+k+1) = tmp;
16   }
17 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

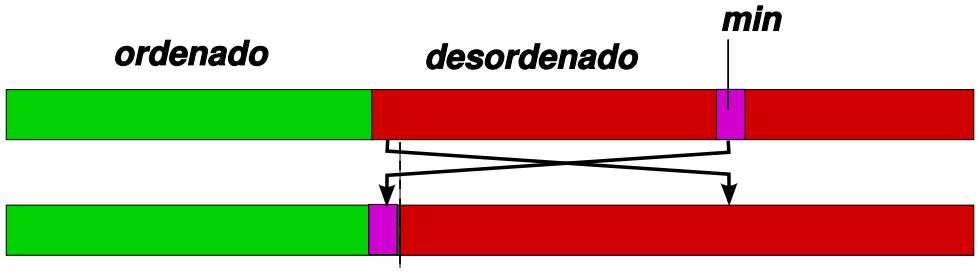
Método de inserción, "insertion-sort" (cont.)

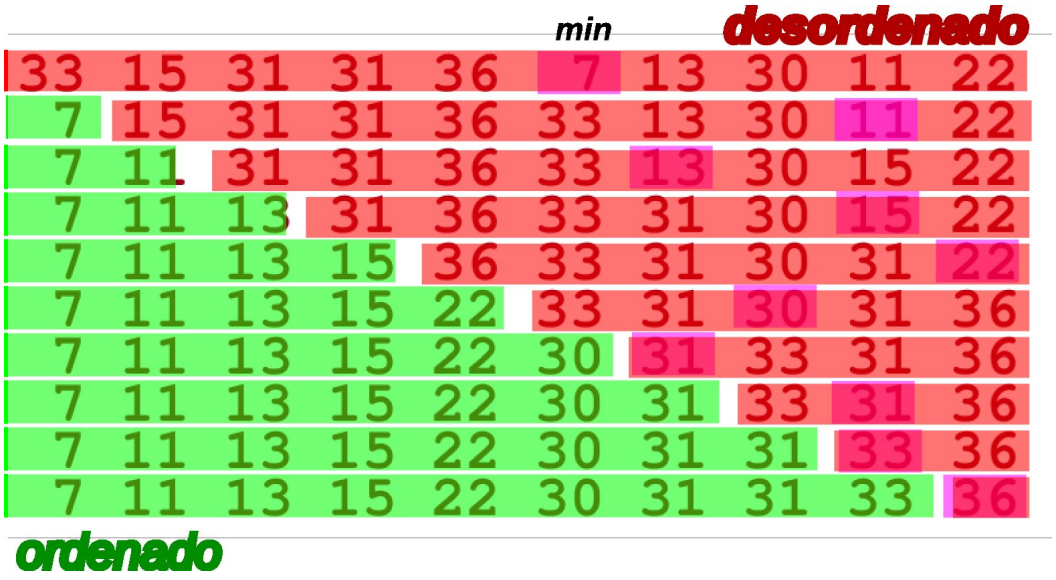
```
1  template<class T> void
2  insertion_sort(typename
3                 std::vector<T>::iterator first,
4                 typename
5                 std::vector<T>::iterator last) {
6    insertion_sort(first,last,less<T>);
7  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de selección, "selection-sort"

- **Hay un doble lazo. En el lazo j se elige el menor del rango $[j, N)$ y se intercambia con el elemento j.**





((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de selección, "selection-sort" (cont.)

```
1   template<class T> void
2   selection_sort(typename
3                  std::vector<T>::iterator first,
4                  typename
5                  std::vector<T>::iterator last,
6                  bool (*comp)(T&,T&)) {
7     int size = last-first;
8     for (int j=0; j<size-1; j++) {
9       typename std::vector<T>::iterator
10        min = first+j,
11        q = min+1;
12      while (q<last) {
13        if (comp(*q,*min)) min = q;
14        q++;
15      }
16      T tmp = *(first+j);
17      *(first+j) = *min;
18      *min = tmp;
19    }
20  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Método de selección, "selection-sort" (cont.)

```
1  template<class T> void
2  selection_sort(typename
3                 std::vector<T>::iterator first,
4                 typename
5                 std::vector<T>::iterator last) {
6    selection_sort(first,last,less<T>);
7  }
```

Tiempos de ejecución

Notación: (mejor/prom/peor)

Método	$T(n)$	**Nro. de intercambios**
Burbuja	$O(n^2)/O(n^2)/O(n^2)$	$0/O(n^2)/O(n^2)$
Inserción	$O(n)/O(n^2)/O(n^2)$	$0/O(n^2)/O(n^2)$
Selección	$O(n^2)/O(n^2)/O(n^2)$	$0/O(n)/O(n)$

La importancia del *número de intercambios* es relativa, ya que cualquier método de ordenamiento se puede llevar a Nro de intercambios $= n$ haciendo *ordenamiento indirecto*, es decir ordenando un vector de *cursores* o *punteros* a los elementos. Típicamente la función retorna una *permutación* que es la que ordena el vector en cuestión, pudiendo tener como *efecto colateral* el ordenamiento (o no) del vector. El usuario entonces puede finalmente aplicar la permutación al vector o a otros vectores relacionados.

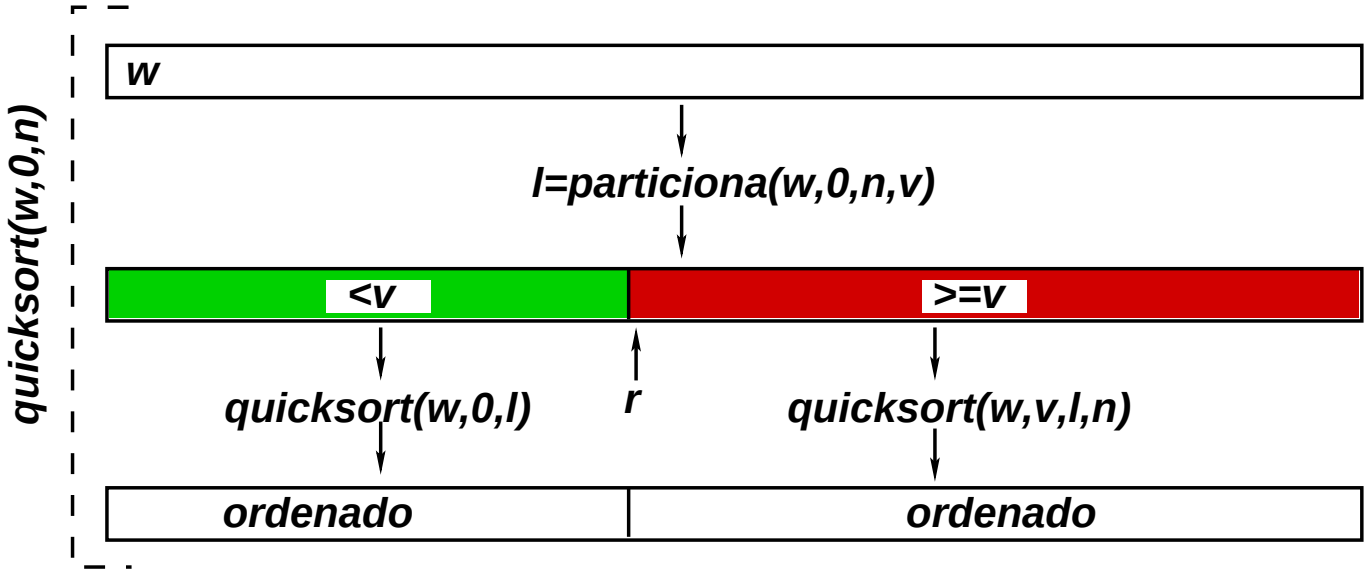
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento rápido (quick-sort)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento rápido (Quick-Sort)

Quick-sort se basa en la estrategia *"dividir para vencer"*. Tomamos un valor *v* (llamado *"pivote"*) y separamos los valores que son mayores o iguales que *v* a la derecha y los menores a la izquierda. Está claro que ahora sólo hace falta ordenar los elementos en cada uno de los subrangos.



Ordenamiento rápido (Quick-Sort) (cont.)

Seudocódigo

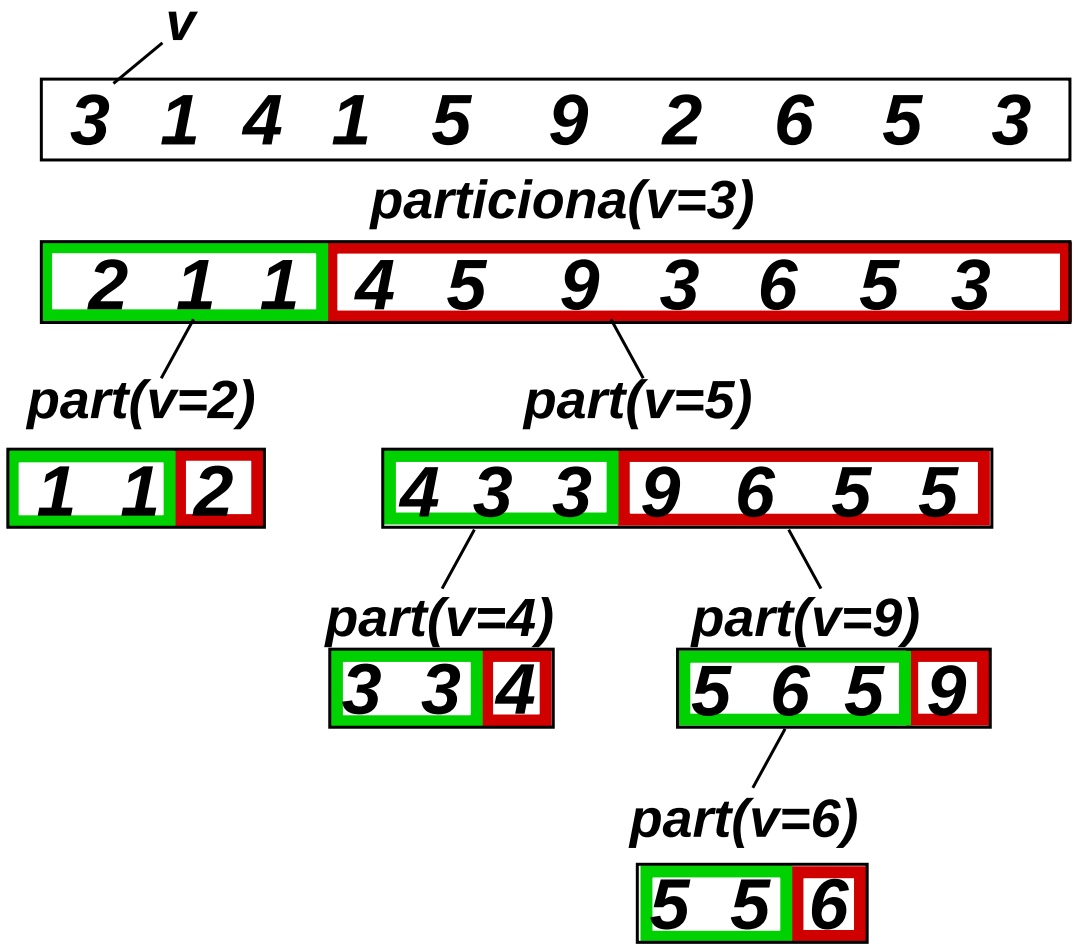
```
1 void quicksort(w,j1,j2) {
2   // Ordena el rango [j1,j2) de 'w'
3   if (n==1) return;
4   // elegir pivote v . . .
5   l = particiona(w,j1,j2,v);
6   quicksort(w,j1,l);
7   quicksort(w,l,j2);
8 }
```

Si $n = j_2 - j_1$ **es la longitud del vector y** $n_1 = l - j_1$ **y** $n_2 = j_2 - l$

$$T(n) = T_{\text{part}}(n) + T(n_1) + T(n_2)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento rápido (Quick-Sort) (cont.)



Temporariamente tomamos como pivote el mayor de los dos primeros distintos. Esto garantiza que al menos cada una de las dos particiones tiene al menos un elemento.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento rápido (Quick-Sort) (cont.)

Si logramos que

- $T_{\text{part}}(n) = cn$ **y que**
- v **sea tal que las longitudes de los subsegmentos estén bien** ***"balanceados"*** **(**$n_1 \approx n_2 \approx n/2$**) entonces**

$$T(n) = T_{\text{part}}(n) + T(n_1) + T(n_2)$$

$$T(n) = cn + 2T(n/2)$$

Ordenamiento rápido (Quick-Sort) (cont.)

Asumiendo $n = 2^p$, $T(1) = d$ **y aplicando recursivamente,**

$$
\begin{aligned}
T(2) &= 2c + 2T(1) = 2c + 2d \\
T(4) &= 4c + 2T(2) = 8c + 4d = 2 \cdot 4c + 4d \\
T(8) &= 8c + 2T(4) = 24c + 8d = 3 \cdot 8c + 8d \\
T(16) &= 16c + 2T(8) = 64c + 16d = 5 \cdot 16c + 16d \\
\vdots &= \quad \vdots \\
T(2^p) &= (p+1)nc + nd
\end{aligned}
$$

Como $p = \log_2 n$

$$T(n) = O(n \log n)$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación

```
1   template<class T>
2   typename std::vector<T>::iterator
3   partition(typename std::vector<T>::iterator first,
4             typename std::vector<T>::iterator last,
5             bool (*comp)(T&,T&),T &pivot) {
6     typename std::vector<T>::iterator
7       l = first,
8       r = last;
9     r--;
10    while (true) {
11      T tmp = *l;
12      *l = *r;
13      *r = tmp;
14      while (comp(*l,pivot)) l++;
15      while (!comp(*r,pivot)) r--;
16      if (l>r) break;
17    }
18    return l;
19  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

```
1   template<class T>
2   int median(typename std::vector<T>::iterator first,
3          typename std::vector<T>::iterator last,
4          std::vector<T> &dif, int k,
5          bool (*comp)(T&,T&)) {
6     typename std::vector<T>::iterator
7       q = first;
8     int ndif=1;
9     dif[0] = *q++;
10    while (q<last) {
11      T val = *q++;
12      int j;
13      for (j=0; j<ndif; j++)
14        // Aca debe compararse por 'equivalente'
15        // es decir usando comp
16        if (!comp(dif[j],val)
17            && !comp(val,dif[j])) break;
18      if (j==ndif) {
19        dif[j] = val;
20        ndif++;
21        if (ndif==k) break;
22      }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23      }
24      typename std::vector<T>::iterator
25        s = dif.begin();
26      bubble_sort(s,s+ndif,comp);
27      return ndif;
28    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Detalles de implementación (cont.)

```
1   template<class T> void
2   quick_sort(typename std::vector<T>::iterator first,
3           typename std::vector<T>::iterator last,
4           bool (*comp)(T&,T&)) {
5     int size = last-first;
6     int max_bub_size = 9;
7     if (size<max_bub_size) {
8       bubble_sort(first,last,comp);
9       return;
10    }
11    if (size<=1) return;
12    int k=3;
13    std::vector<T> dif(k);
14    int ndif = median(first, last, dif, k, comp);
15    if (ndif==1) return;
16    T pivot = dif[ndif/2];
17    typename std::vector<T>::iterator l;
18    l = partition(first,last,comp,pivot);
19    quick_sort(first,l,comp);
20    quick_sort(l,last,comp);
21  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22
23  template<class T> void
24  quick_sort(typename std::vector<T>::iterator first,
25             typename std::vector<T>::iterator last) {
26    quick_sort(first,last,less<T>);
27  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Peor caso

Si el particionamiento es muy desbalanceado $n1 = 1$ y $n2 = n - 1$ entonces

$$T(n) = cn + T(1) + T(n-1)$$

$$T(2) = 2c + 2d$$

$$T(3) = 3c + d + (2c + 2d) = 5c + 3d$$

$$T(4) = 4c + d + (5c + 3d) = 9c + 4d$$

$$T(5) = 5c + d + (9c + 4d) = 14c + 5d$$

$$\vdots \quad = \quad \vdots$$

$$T(n) = \left(\frac{n(n+1)}{2} - 2\right) c + nd = O(n^2)$$

En particular esto ocurre si el vector esta inicialmente ordenado u ordenado en forma inversa.

Quick-sort. Observaciones

- **En el caso promedio puede demostrarse que el orden se mantiene $O(n \log n)$.**
- **Mejoras incluyen un mejor cálculo del pivote (mediana? mediana de k primeros distintos?)**
- **Para longitudes de secuencias cambiar a un método simple (burbuja, etc...)**

(lanzar video sorting.avi)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por montículos (heap-sort)

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por montículos. (Heapsort)

- Supongamos que podemos implementar para un cierto contenedor una estructura que permite implementar las funciones `insert(x)` `p=min(S)` y `x=*p` y `erase(p)` en tiempo $O(\log n)$. Entonces podemos ordenar un contenedor (como una lista `L`, por ejemplo)

```
1  // Pone todos los elementos en S
2  while (!L.empty()) {
3    x = *L.begin();
4    S.insert(x);
5    L.erase(L.begin());
6  }
7  // Saca los elementos de S usando 'min'
8  while (!S.empty()) {
9    p = min(S);
10   L.push(L.end(),*p);
11   S.erase(p);
12 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por montículos. (Heapsort) (cont.)

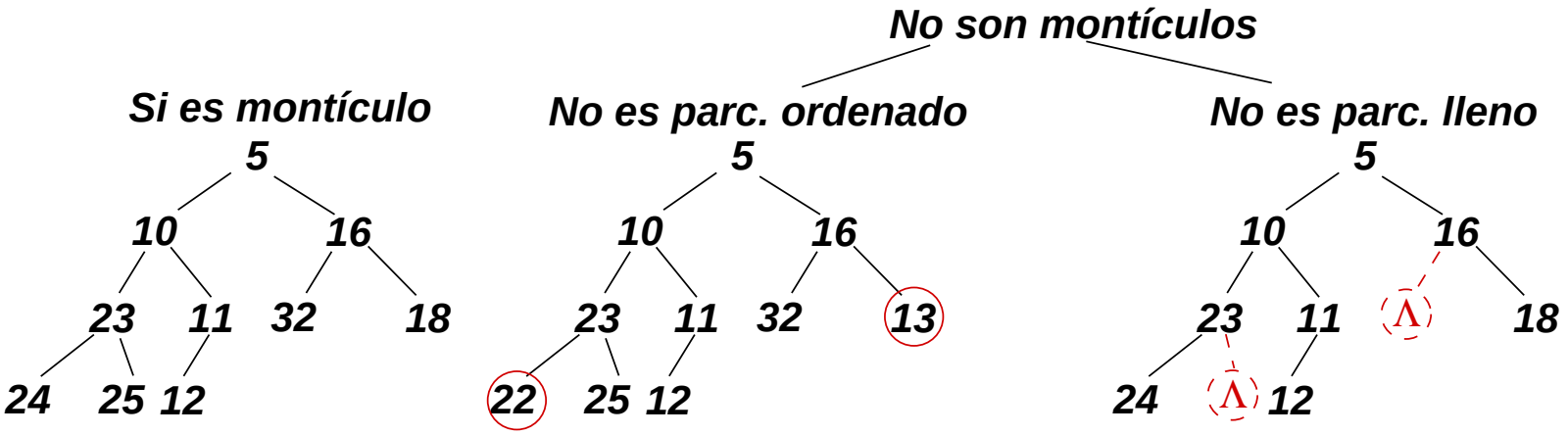
- **Como cada ejecución del lazo es $O(\log n)$ (en realidad va disminuyendo ya que el número de elementos en S va disminuyendo) tenemos $T(n) = O(n \log n)$.**
- **Ej: árbol binario de búsqueda (si se mantiene bien balanceado).**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Montículo

Un montículo es un árbol binario que satisface las siguientes condiciones.

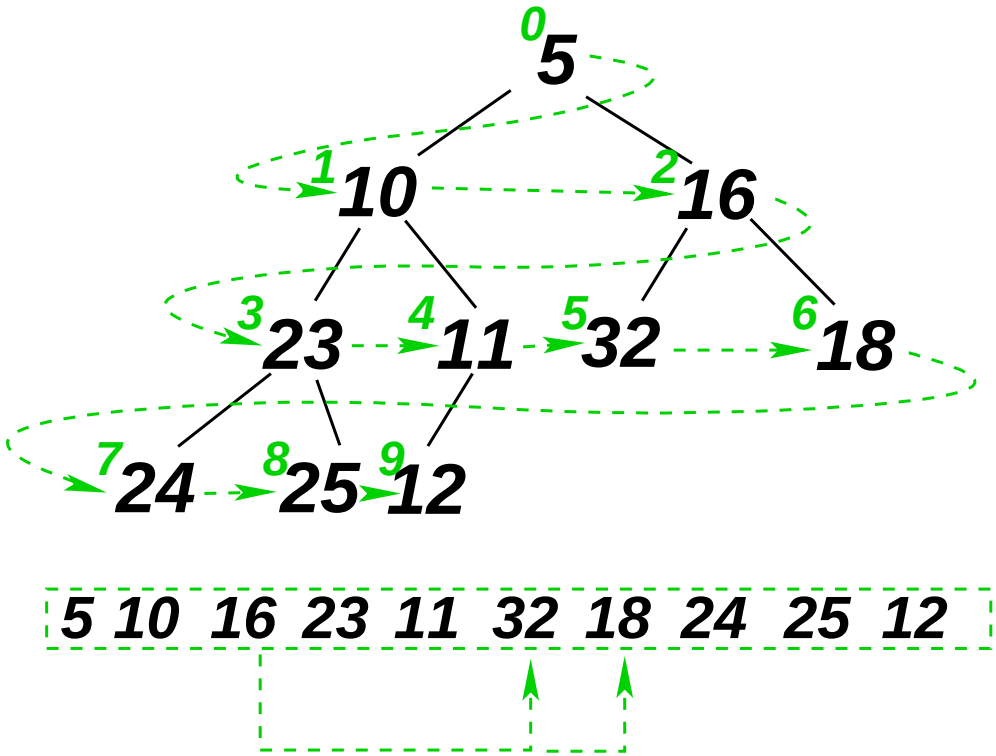
- **Es *"parcialmente ordenado"*, es decir el padre es siempre menor o igual que sus dos hijos.**
- **Es parcialmente lleno: Todos los niveles están ocupados, menos el último nivel, en el cual están ocupados todos los lugares más a la izquierda.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Montículo (cont.)

- **La condición de que sea parcialmente ordenado implica que el mínimo está siempre en la raíz.**
- **La condición de parcialmente lleno permite implementarlo eficientemente en un vector.**



hijo izquierdo(j) $= 2j + 1,$

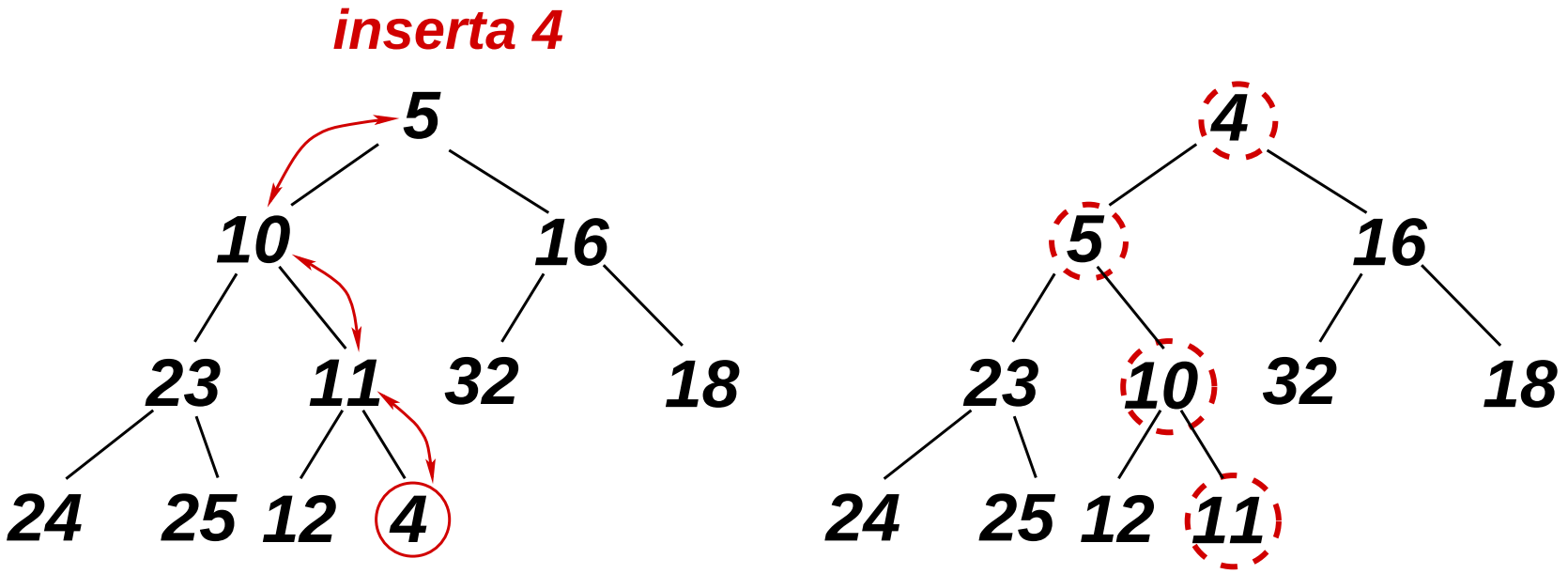
hijo derecho(j) $= 2j + 2,$

padre(j) $= \mathrm{floor}((j-1)/2).$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Inserción en montículo

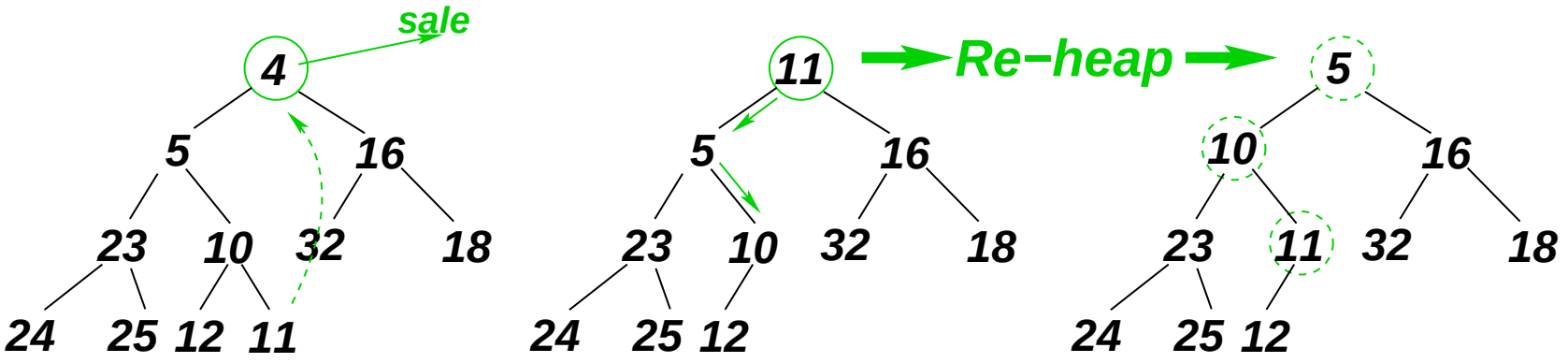
- **Poner el elemento en la primera posición libre (posición más a la izquierda del último nivel semilleno)**
- **Subir intercambiando con el padre, si es menor.**
- **Costo:** $O(\log_2 n)$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Eliminar el mínimo. Re-heap.

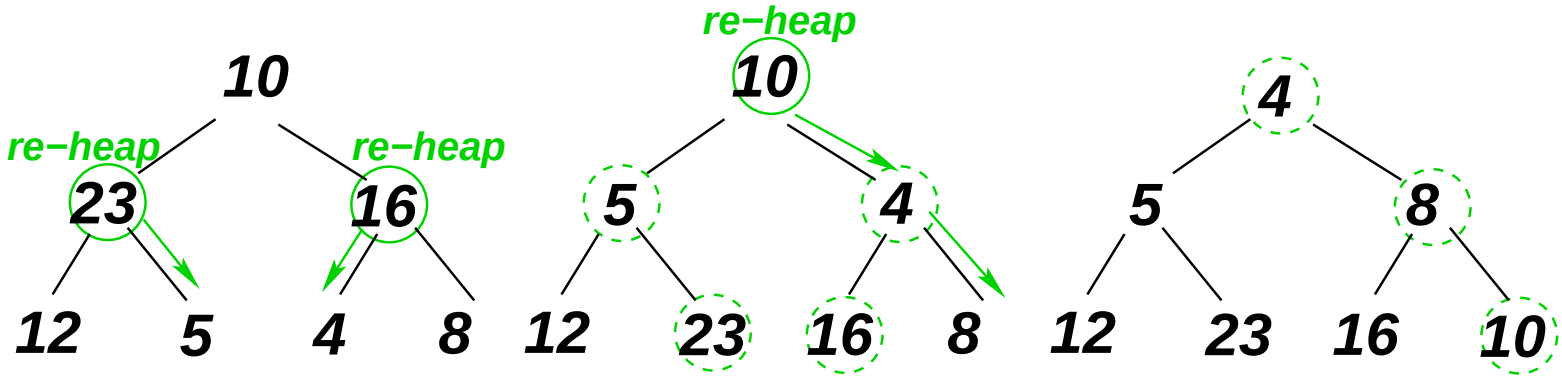
- **Sacamos el mínimo (el elemento de la raíz) y subimos el último elemento (el más a la derecha del último nivel)**
- **Lo único que no se verifica es la condición de parcialmente ordenado en el nodo raíz.**
- **La longitud del vector se acorta en uno.**
- **Aplicamos el procedimiento *"re-heap"* (*"rehacer montículo"*) que va *"bajando"* el elemento siempre por el hijo menor (mientras el padre sea mayor). Costo:** $O(\log_2 n)$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Make-heap.

- **Tenemos un vector completamente desordenado y queremos hacer un montículo.**
- **Aplicar re-heap a todos los nodos interiores desde abajo hacia arriba.**
- **Costo:** $O(n \log_2 n)$ **(En realidad es** $O(n)$**).**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación

```
1   template<class T> void
2   re_heap(typename std::vector<T>::iterator first,
3           typename std::vector<T>::iterator last,
4           bool (*comp)(T&,T&),int j=0) {
5     int size = (last-first);
6     T tmp;
7     while (true) {
8       typename std::vector<T>::iterator
9         higher,
10        father = first + j,
11        l = first + 2*j+1,
12        r = l + 1;
13      if (l>=last) break;
14      if (r<last)
15        higher = (comp(*l,*r) ? r : l);
16      else higher = l;
17      if (comp(*father,*higher)) {
18        tmp = *higher;
19        *higher = *father;
20        *father = tmp;
21      }
22      j = higher - first;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23      }
24    }
25
26    template<class T> void
27    make_heap(typename std::vector<T>::iterator first,
28              typename std::vector<T>::iterator last,
29                 bool (*comp)(T&,T&)) {
30      int size = (last-first);
31      for (int j=size/2-1; j>=0; j--)
32        re_heap(first,last,comp,j);
33    }
34
35    template<class T> void
36    heap_sort(typename std::vector<T>::iterator first,
37              typename std::vector<T>::iterator last,
38              bool (*comp)(T&,T&)) {
39      make_heap(first,last,comp);
40      typename std::vector<T>::iterator
41        heap_last = last;
42      T tmp;
43      while (heap_last>first) {
44        heap_last--;
45        tmp = *first;
46        *first = *heap_last;
47        *heap_last = tmp;
```

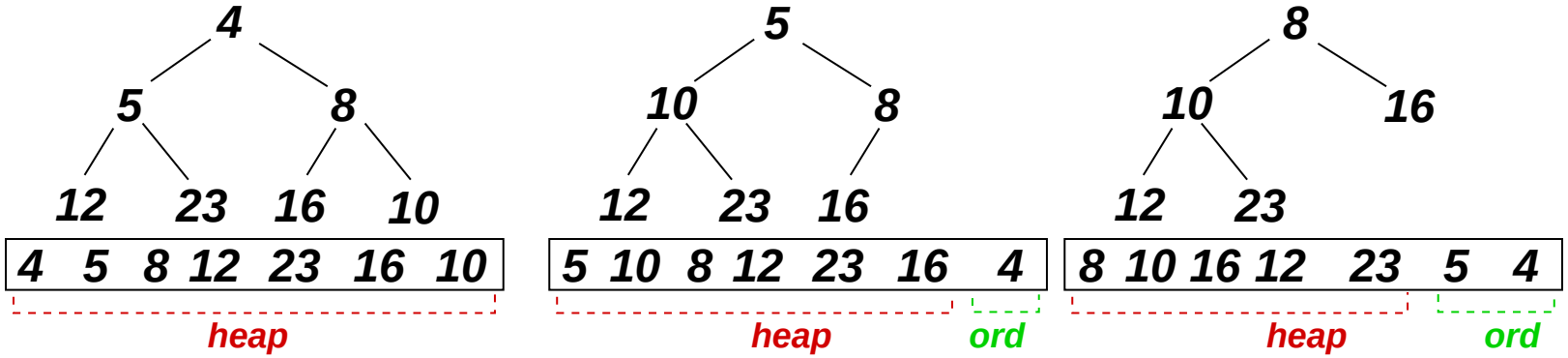

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
48      re_heap(first,heap_last,comp);
49    }
50  }
51
52  template<class T> void
53  heap_sort(typename std::vector<T>::iterator first,
54            typename std::vector<T>::iterator last) {
55    heap_sort(first,last,less<T>);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Implementación in-place en un vector

- **Se aplica primero *make-heap***
- **Se va eliminando el mínimo y poniéndolo al final y aplicando *re-heap* para volver a hacer el montículo.**
- **Finalmente el vector queda ordenado de mayor a menor.**
- **El vector se invierte ($O(n)$).**
- **Otra posibilidad es usar un *"montículo maximal"* (el padre es mayor o igual que los hijos). En cuyo caso no hace falta la inversión final.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Comparación de heap-sort y quick-sort

- **Quicksort es $O(n \log n)$ en el caso promedio, $O(n^2)$ en el peor caso.**
- **Heap-sort es $O(n \log n)$ *siempre*.**
- **En el caso promedio quick-sort es más rápido que heap-sort por un factor constante.**
- **Podríamos hacer quick-sort siempre $O(n \log n)$ desordenando primero el vector (random-shuffle), pero en ese caso perdería contra heap-sort.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad de los diferentes esquemas

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad

Recordemos que si `comp(x,y)=comp(y,x)=false` decimos que `x,y` son *equivalentes*. La condición pedida para la función de comparación `bool comp(x,y)` es que sea un `preorden total (strict weak ordering)` es decir que puede haber elementos equivalentes entre sí pero que no son iguales. Por ejemplo si se trata de la comparación por valor absoluto: `comp(x,y) = abs(x)<abs(y)`, entonces 2 y -2 son equivalentes entre sí pero no son iguales.

Si ordenamos los elementos `(1,-2,-3,2)` puede dar `(1,-2,2,-3)` ó `(1,2,-2,-3)` y ambos resultados son válidos.

Si los elementos equivalentes entre sí quedan en el mismo orden que en la secuencia original, entonces decimos que el algoritmo de ordenamiento es *estable*. En el caso anterior el resultado `(1,-2,2,-3)` es estable mientras que `(1,2,-2,-3)` no lo es.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad (cont.)

Notemos que en el caso que la relación de orden sea un *orden total (total ordering)*, entonces si dos elementos son equivalentes implica que son iguales, y por lo tanto no existe esta ambigüedad, es decir *la estabilidad del algoritmo de ordenamiento es irrelevante*.

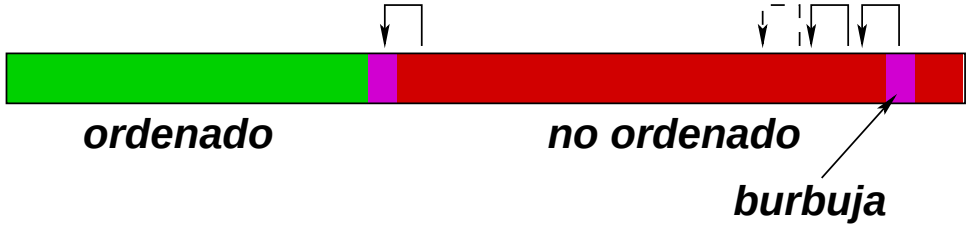
El interés por que un algoritmo sea estable puede ser

- **Si el algoritmo es estable entonces no existe ambigüedad en el resultado, es decir dada una secuencia de entrada, *la secuencia ordenada resultante es la misma siempre*.**
- **Si se ordenan registros por varias *claves*, por ejemplo registros de empleados primero por *nombre de empleado* y después por *nombre de empresa*, entonces el último ordenamiento no *destruye* el ordenamiento previo, es decir los registros quedan ordenados por empresa, y dentro del rango de cada empresa quedan ordenados por nombre de empleado.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad (cont.)

Una forma de verificar si un algoritmo de ordenamiento es estable o no es controlando que en ningún momento se intercambien *elementos equivalentes*. En el caso de *bubble-sort* el intercambio sólo se realiza si `*(first+k)` es *estrictamente menor* que `*(first+k-1)` y por lo tanto *el intercambio nunca viola la estabilidad*.

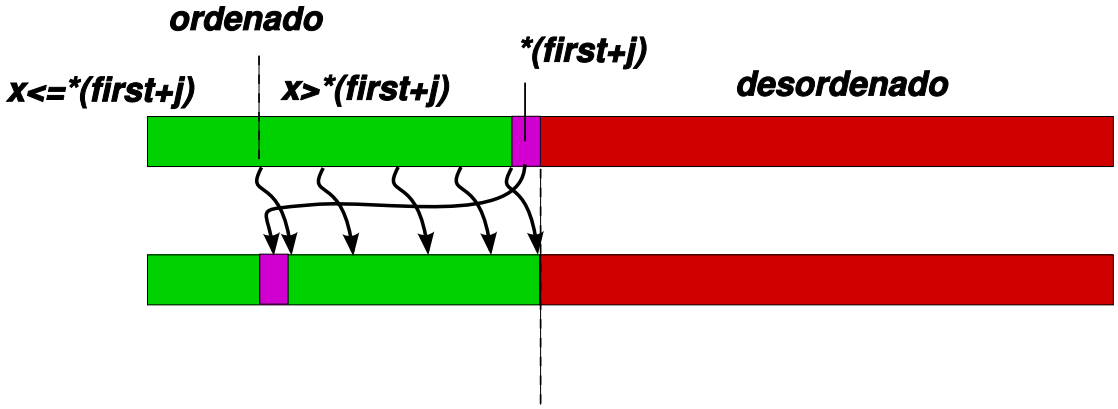



```
1       for (int k=size-1; k>j; k--) {
2         if (comp(*(first+k),*(first+k-1))) {
3           T tmp = *(first+k-1);
4           *(first+k-1) = *(first+k);
5           *(first+k) = tmp;
6         }
7       }
```

por lo tanto *bubble -sort SI es estable*.

Estabilidad de inserción

El elemento `*(first+j)` es intercambiado con todo el rango `[first+k,first+j)` que son elementos *estrictamente mayores* que `*(first+j)`.



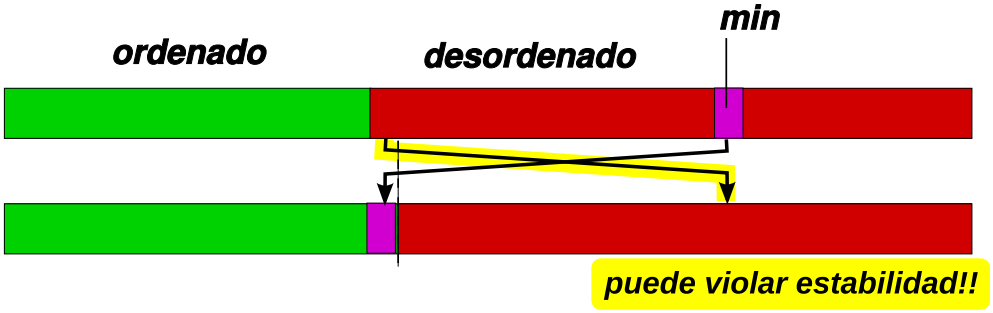

```
1     for (int j=1; j<size; j++) {
2       T tmp = *(first+j);
3       int k=j-1;
4       while (comp(tmp,*(first+k))) {
5         *(first+k+1) = *(first+k);
6         if (--k < 0) break;
7       }
8       *(first+k+1) = tmp;
9     }
```

por lo tanto *insertion-sort SI es estable*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad de selección

El elemento `*(first+j)`, que va a ir a la posición `min`, puede estar cambiando de posición relativa con elementos equivalentes en el rango `(first+j,min)`, violando la estabilidad.



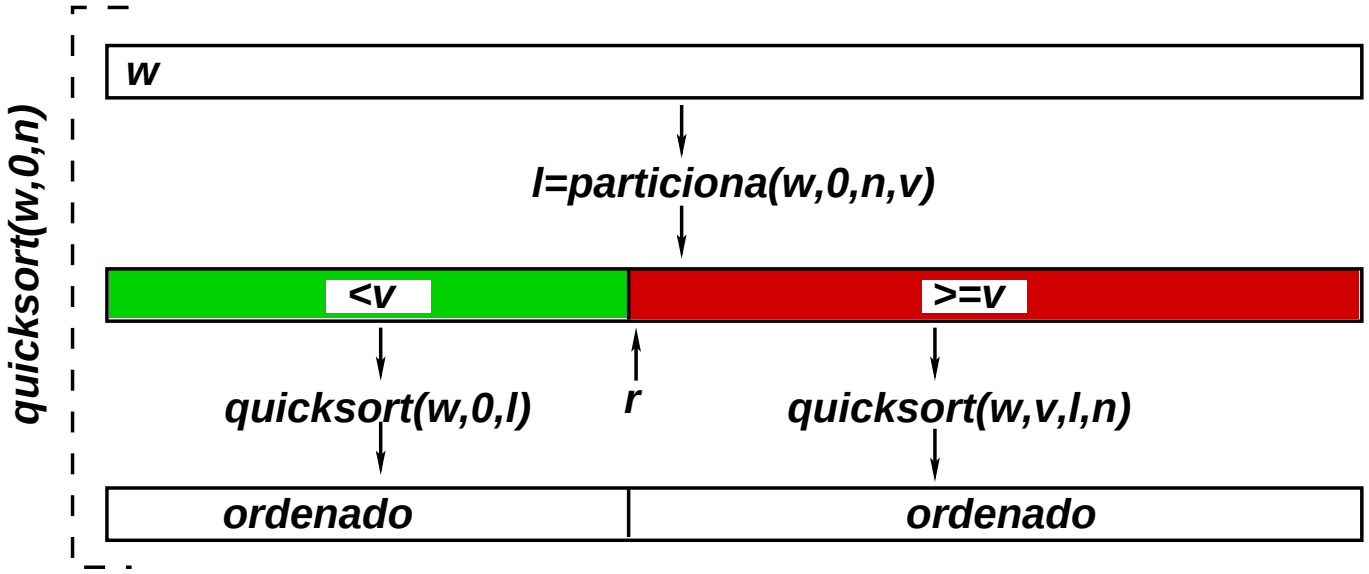

```
1     for (int j=0; j<size-1; j++) {
2       typename std::vector<T>::iterator
3         min = first+j,
4         q = min+1;
5       while (q<last) {
6         if (comp(*q,*min)) min = q;
7         q++;
8       }
9       T tmp = *(first+j);
10      *(first+j) = *min;
11      *min = tmp;
12    }
```

por lo tanto *selection-sort NO es estable*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad de quick-sort

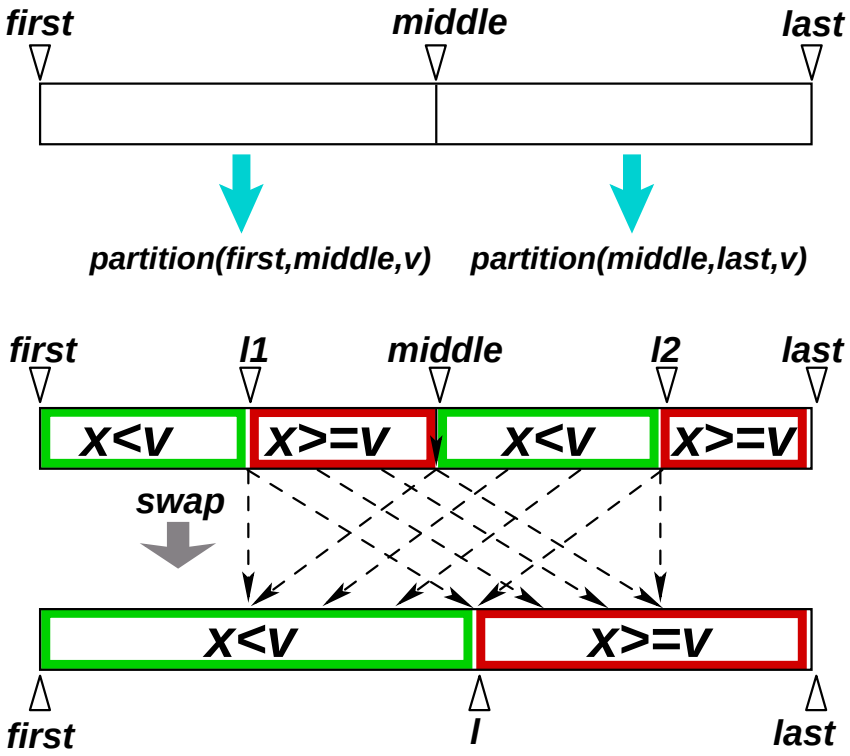
Quick-sort es estable si el algoritmo de particionamiento es estable, ya que una vez que hacemos la partición los elementos en cada subsecuencia no se mueven más. Tal cual como está implementado aquí, el algoritmo de partición no es estable.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad de quick-sort (cont.)

Se puede implementar un algoritmo de partición estable recursivo. Dividimos el rango `[first,last)` por el punto medio `middle`. Aplicamos recursivamente `partition()` a cada una de los subrangos y los intercambiamos (*"swap"*). Si el particionamiento de ambos subrangos fue estable y el swap mantiene el orden relativo, entonces la particioón será estable, ya que los elementos de `[middle,l2)` son estrictamente mayores que los de `[l1,middle)`.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Estabilidad de quick-sort (cont.)

- **Notar que tenemos dos lazos recursivos anidados: un lazo externo que hace el quick-sort y uno interno para cada partición.**
- **Si el tiempo de swap es $O(n)$, entonces se puede hacer un análisis similar al de quick-sort (para el caso promedio) y se llega a la conclusión que el tiempo de ejecución de esta versión del algoritmo de partición es $O(n \log n)$ *siempre*. (Aquí elegimos el punto medio `middle` siempre en la mitad mientras que en quick-sort las subsecuencias pueden llegar a estar desbalanceadas).**
- **`swap()` se puede implementar en tiempo $O(n)$ *in-place* (ver apuntes). (Si relajamos la condición de ser *in-place* es trivial).**
- **Al aumentar el tiempo de ejecución del particionamiento, el tiempo de ejecución de quick-sort pasa a ser $O(n(\log n)^2)$ (ver apuntes).**

Estabilidad de heap-sort

Heap-sort es intrínsecamente no-estable ya que al intercambiar elementos en el montículo no hay forma de garantizar que no se pasa por encima de elementos equivalentes.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

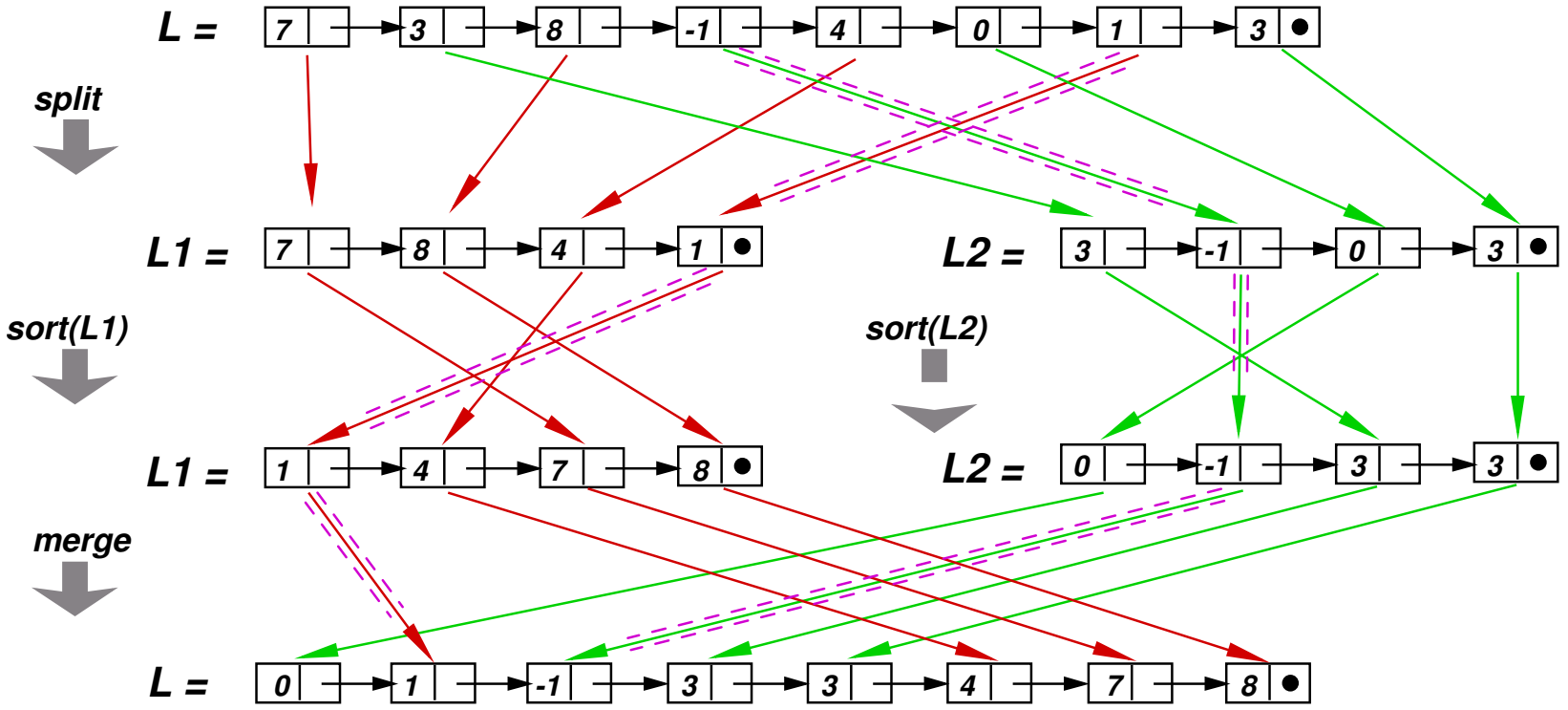
Ordenamiento por fusión

Ordenamiento por fusión

Es conceptualmente, uno de los algoritmos rápidos más simples de comprender (a.k.a. *"intercalamiento"* ó *"merge-sort"*). Pensemos primero en listas. La estrategia es típica de *"dividir para vencer"* e intrínsecamente recursiva. La lista se divide (*"split"*) en dos sublistas del mismo tamaño y se aplica recursivamente `merge_sort()` a cada una de las listas. Luego estas se concatenan (también *"fusionan"* o *"merge"*) en `L` manteniendo el orden, como en `set_union()` para conjuntos por listas ordenadas. Si la división se hace en forma balanceada y las operaciones de división y concantenación se pueden lograr en tiempo $O(n)$ entonces el análisis de costo es similar al de quick-sort en el caso promedio y finalmente se obtiene un tiempo $O(n \log n)$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por fusión (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por fusión (cont.)

El tiempo de ejecución satisface una ecuación recursiva como para *quick-sort*:

$$T(n) = T(\textbf{split+merge}) + T(n_1) + T(n_2)$$

Ya hemos visto que el algoritmo de concatenación para listas ordenadas es $O(n)$ y para dividir la lista en dos de igual tamaño simplemente podemos ir tomando un elemento de `L` e ir poniéndolo alternadamente en `L1` y `L2`, lo cual es $O(n)$. A esta forma de separar la lista en dos de igual tamaño lo llamamos *"splitting par/impar"*.

De manera que ciertamente es $T(\textbf{split+merge}) = O(n)$ y por lo tanto $T(\textbf{merge-sort}) = O(n \log n)$.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por fusión (cont.)

Es merge-sort in place? **El concepto de si el ordenamiento es *in-place* o no cambia un poco para listas. Si bien usamos contenedores auxiliares (las listas `L1` y `L2`), la cantidad total de celdas en juego es siempre n, si tomamos la precaución de ir eliminando las celdas de `L` a medidas que insertamos los elementos en las listas auxiliares, y *viceversa* al hacer la fusión. De manera que podemos decir que merge-sort es in-place.**

Merge-sort es el algoritmo de elección para listas: **Es simple, $O(n \log n)$ en el peor caso, y es *in-place*, mientras que cualquiera de los otros algoritmos rápidos como *quick-sort* y *heap-sort* se vuelven cuadráticos (o peor aún) al querer adaptarlos a listas, debido a la falta de iteradores de acceso aleatorio. También merge-sort es la base de los algoritmos para *ordenamiento externo*.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por fusión (cont.)

```
1   template<class T> void
2   merge_sort(std::list<T> &L,bool (*comp)(T&,T&)) {
3     std::list<T> L1,L2;
4     list<T>::iterator p = L.begin();
5     if (p==L.end() || ++p==L.end()) return;
6     bool flag = true;
7     while (!L.empty()) {
8       std::list<T> &LL = (flag ? L1 : L2);
9       LL.insert(LL.end(),*L.begin());
10      L.erase(L.begin());
11      flag = !flag;
12    }
13
14    merge_sort(L1,comp);
15    merge_sort(L2,comp);
16
17    typename std::list<T>::iterator
18      p1 = L1.begin(),
19      p2 = L2.begin();
20    while (!L1.empty() && !L2.empty()) {
21      std::list<T> &LL =
22        (comp(*L2.begin(),*L1.begin()) ? L2 : L1);
```

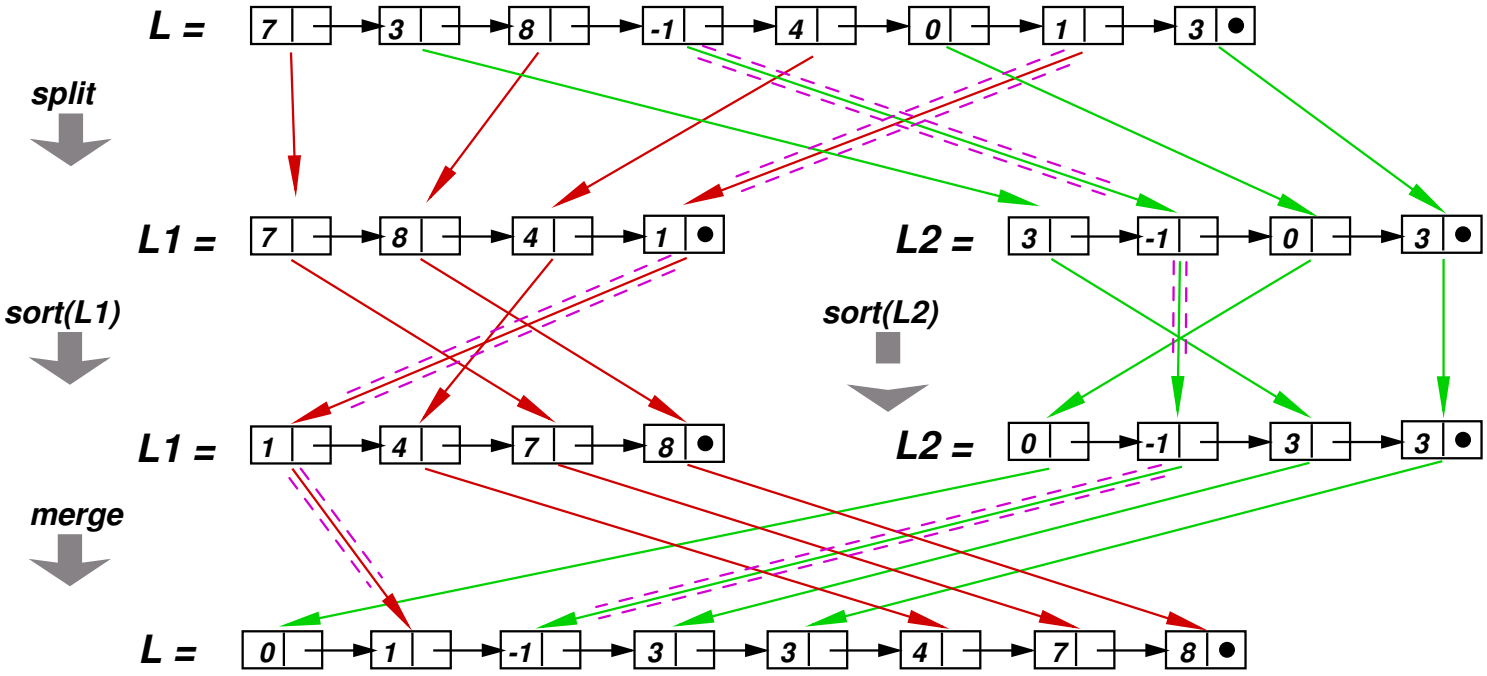

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23        L.insert(L.end(),*LL.begin());
24        LL.erase(LL.begin());
25      }
26      while (!L1.empty()) {
27        L.insert(L.end(),*L1.begin());
28        L1.erase(L1.begin());
29      }
30      while (!L2.empty()) {
31        L.insert(L.end(),*L2.begin());
32        L2.erase(L2.begin());
33      }
34    }
35
36    template<class T>
37    void merge_sort(std::list<T> &L) {
38      merge_sort(L,less<T>);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Merge-sort estable

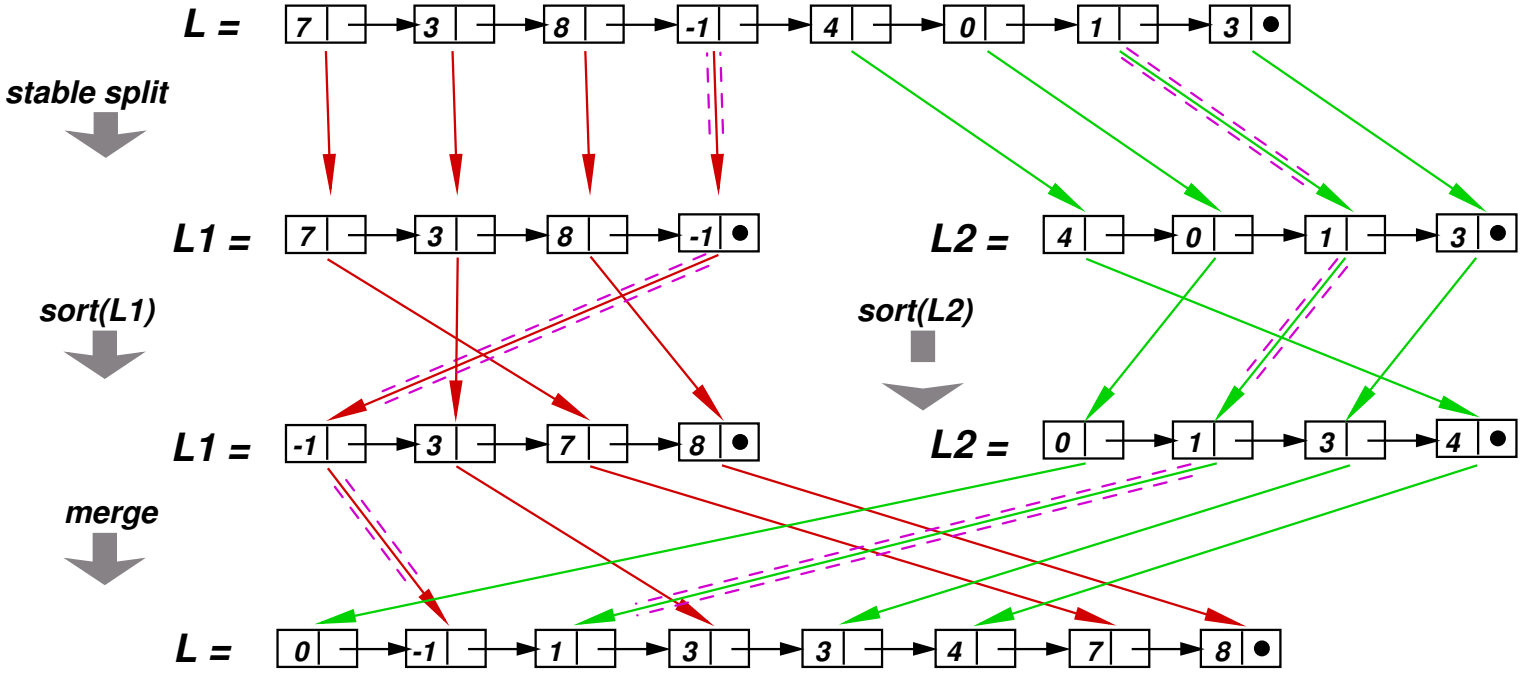
Es fácil implementar la etapa de fusión (*"merge"*) en forma estable. Sin embargo, así como está implementado los elementos equivalentes se mezclan inevitablemente en la etapa de *split*.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Merge-sort estable (cont.)

Para que *merge-sort* se debe hacer el *split* en forma estable mandando los primeros $n_1 = \text{floor}(n/2)$ a `L1` y los demás ($n_2 = n - \text{floor}(n/2) = \text{ceil}(n/2)$) a `L2`. Después, al hacer el *merge*, cuando ambos elementos son equivalentes se elige el de la lista `L1`.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Versión estable de split

```
1     int size = L.size();
2     if (size==1) return;
3     std::list<T> L1,L2;
4     int n1 = size/2;
5     int n2 = size-n1;
6     for (int j=0; j<n1; j++) {
7       L1.insert(L1.end(),*L.begin());
8       L.erase(L.begin());
9     }
10    for (int j=0; j<n2; j++) {
11      L2.insert(L2.end(),*L.begin());
12      L.erase(L.begin());
13    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento por fusión para vectores

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Merge-sort para vectores

Es muy simple implementar una versión de merge-sort para vectores, si se usa un vector auxiliar (es decir que el algoritmo no es *in-place*). El proceso es igual que para listas, pero al momento de hacer la fusión, esta se hace sobre el vector auxiliar. Por eficiencia, este vector debe ser en principio tan largo como el vector original, es creado en una función *"wrapper"* auxiliar y pasado siempre por referencia.

- **Para vectores no es necesario hacer explícitamente el *split* ya que basta con pasar los extremos de los intervalos. Es decir, la operación de split es aquí un simple cálculo del iterator `middle`, que es $O(1)$.**
- **El vector auxiliar `tmp` se crea en el wrapper. Este vector auxiliar es pasado a una función recursiva `merge_sort(first,last,tmp,comp)`.**
- **El algoritmo de fusión es igual que para listas, pero los elementos se van copiando a elementos del vector `tmp` (el iterator `q`).**
- **El vector ordenado es recopiado de vuelta en `[first,last)`.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Merge-sort para vectores (cont.)

```
1  template<class T> void
2  merge_sort(typename std::vector<T>::iterator first,
3             typename std::vector<T>::iterator last,
4             typename std::vector<T> &tmp,
5             bool (*comp)(T&,T&)) {
6    int
7      n = last-first;
8    if (n==1) return;
9    int n1 = n/2, n2 = n-n1;
10   typename std::vector<T>::iterator
11     middle = first+n1,
12     q = tmp.begin(),
13     q1 = first,
14     q2 = first+n1;
15
16   merge_sort(first,middle,tmp,comp);
17   merge_sort(first+n1,last,tmp,comp);
18
19   while (q1!=middle && q2!=last) {
20     if (comp(*q2,*q1)) *q++ = *q2++;
21     else *q++ = *q1++;
22   }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
23      while (q1!=middle) *q++ = *q1++;
24      while (q2!=last) *q++ = *q2++;
25
26      q1=first;
27      q = tmp.begin();
28      for (int j=0; j<n; j++) *q1++ = *q++;
29    }
30
31    template<class T> void
32    merge_sort(typename std::vector<T>::iterator first,
33               typename std::vector<T>::iterator last,
34                  bool (*comp)(T&,T&)) {
35      std::vector<T> tmp(last-first);
36      merge_sort(first,last,tmp,comp);
37    }
38
39    template<class T> void
40    merge_sort(typename std::vector<T>::iterator first,
41               typename std::vector<T>::iterator last) {
42      merge_sort(first,last,less<T>);
43    }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Merge-sort para vectores (cont.)

Esta implementación es estable, $O(n \log n)$ en el peor caso, pero no es in-place. Si relajamos la condición de estabilidad, entonces podemos usar el algoritmo de intercalación discutido en el apunte. Ese algoritmo es $O(n)$ y no es in-place, pero requiere de menos memoria adicional, $O(\sqrt{n})$ en el caso promedio, $O(n)$ en el peor caso, en comparación con el algoritmo descripto aquí que que requiere $O(n)$ siempre,

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort

- **Aquí el punto más importante es minimizar las veces que se accede a un dato en particular. Además se debe tratar de acceder a todos los datos en forma secuencial $a_j, a_{j+1}, a_{j+2}\ldots$, y no en forma aleatoria, ya que la lectura en forma secuencial es más eficiente ya que maximiza la probabilidad de utilizar en forma eficiente los *buffers* de acceso a disco.**
- **Por lo tanto, los algoritmos de ordenamiento apropiados para vectores no suelen ser apropiados para ordenamiento externo ya que involucran acceder aleatoriamente a los elementos, mientras que los algoritmos apropiados para listas (como *merge-sort*) si son apropiados ya que tienden a acceder a los elementos en forma secuencial.**

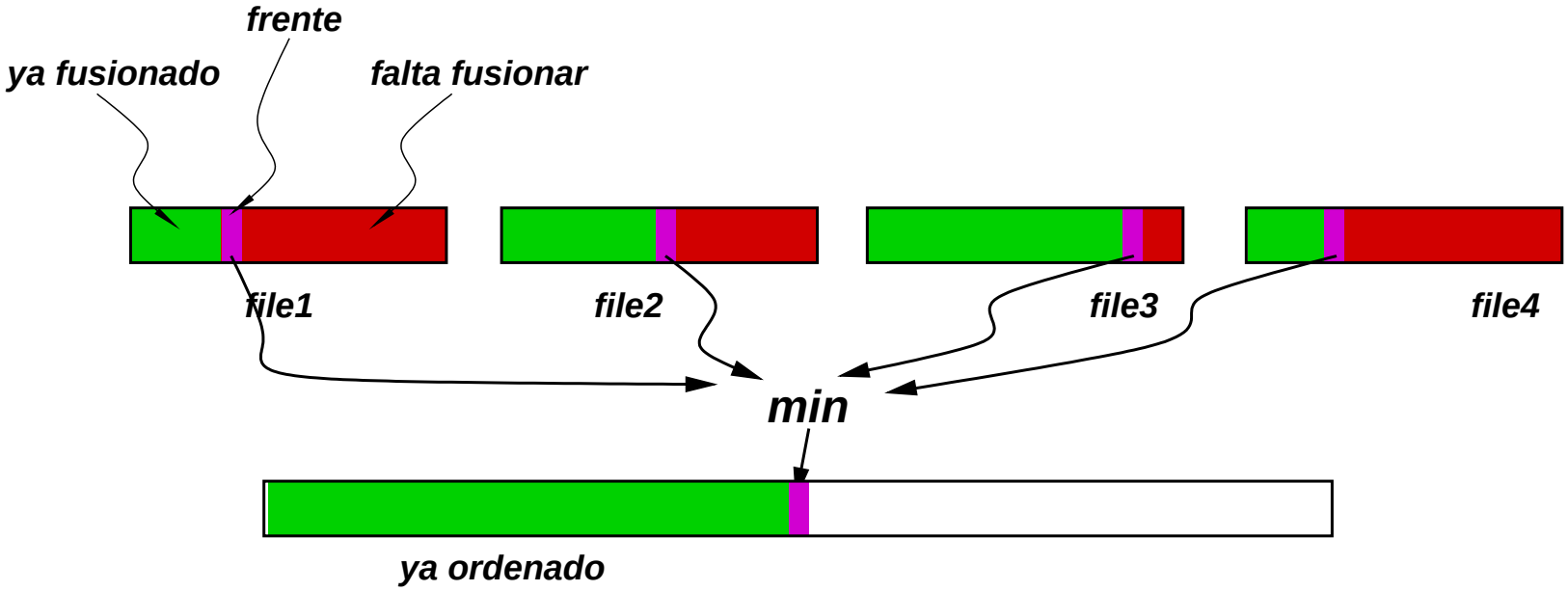
Ordenamiento externo con merge-sort (cont.)

- **Dividir a los datos a ordenar en k archivos, de tal manera que el tamaño total de los datos en cada archivo se puede ordenar en forma interna.**
- **Ordenar los elementos de cada archivo (en forma interna).**
- **Aplicar un algoritmo de fusión a los archivos como si fueran secuencias. Este algoritmo, llamado *"K-way"* merge, es la generalización del algoritmo de fusión para listas pero involucra un número de secuencias $k >= 2$.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- **Inicializar un frente que contiene los elementos más bajos de cada archivo.**
- **En todo momento vamos tomando el menor elemento del frente y lo insertamos a la cola en el vector ordenado. El siguiente elemento de la secuencia donde se encontró el mínimo pasa a reemplazarlo.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

Si el número máximo de elementos que se pueden ordenar en forma interna es m, entonces el número de archivos es $k = n/m$ (asumimos que n es múltiplo de m). Cada vez que insertamos un elemento debemos calcular el mínimo ($O(k)$) y el resto es $O(1)$, de manera que tenemos para la fusión $O(kn) = O(n^2/m)$ operaciones. Además para el ordenamiento inicial tenemos $kO(m \log m) = O(n \log m)$. (Asumimos que usamos un algoritmo $O(m \log m)$ para el ordenamiento interno.)

Si k es muy grande puede usarse una cola de prioridad para extraer el mínimo, de manera que llegaríamos a una estimación $O(n \log(n/m)) = O(n \log n)$.

Ordenamiento externo con merge-sort (cont.)

- `external_sort(char *file_in,char *file_out)` ordena los enteros en un archivo `file_in` y los deja ordenados en `file_out`.

```
1  int external_sort(const char *file_in,
2                    const char *file_out) {
3    FILE *fid = fopen(file_in,"r");
4    int file_indx = 0;
5    vector<int> v;
6    while (true) {
7      int w;
8      int nread = fscanf(fid,"%d",&w);
9      if (nread==EOF) break;
10     v.push_back(w);
11     if (v.size()>=MAX)
12       flush(v,file_indx++);
13   }
14   if (v.size()) flush(v,file_indx++);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- La función `flush(vector<int> &v, int fil_indx)` guarda los elementos del vector `v` en un archivo temporario cuyo nombre es generado a partir de `file_indx` (por ejemplo, si `file_indx=12` entonces se usará `tmp-file-12.dat`).

```
1  void flush(vector<int> &v,int file_indx) {
2    char tmp_file[100];
3    sort(v.begin(),v.end());
4    sprintf(tmp_file,"tmp-file-%d.dat",file_indx);
5    FILE *tmp = fopen(tmp_file,"w");
6    for (int j=0; j<v.size(); j++)
7      fprintf(tmp,"%d\n",v[j]);
8    fclose(tmp);
9    v.clear();
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- `files` es un vector de *punteros a archivos*. A medida que los archivos se van terminando los ponemos a `NULL`.
- `front` es un vector con los elementos mínimos actuales en cada archivo.

```
1  int nfiles = file_indx;
2  vector<FILE *> files(nfiles,(FILE*)(NULL));
3  vector<int> front(nfiles);
4  int w;
5  char tmp_file[100];
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- Empezamos abriendo todos los archivos en `files` y tomando el mínimo (es decir el primer elemento) de cada uno en `front`.

```
1  for (int j=0; j<nfiles; j++) {
2    sprintf(tmp_file,"tmp-file-%d.dat",j);
3    files[j] = fopen(tmp_file,"r");
4    int nread = fscanf(files[j],"%d",&w);
5    assert(nread==1);
6    front[j] = w;
7  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

```
1   // K-way merge
2   FILE *out = fopen(file_out,"w");
3   while (1) {
4     int jmin=-1;
5     for (int j=0; j<nfiles; j++) {
6       if (!files[j]) continue;
7       if (jmin<0 || front[j]<front[jmin])
8         jmin = j;
9     }
10    if (jmin<0) break;
11    fprintf(out,"%d\n",front[jmin]);
12    int nread = fscanf(files[jmin],
13                       "%d",&front[jmin]);
14    if (nread!=1) {
15      fclose(files[jmin]);
16      files[jmin]=NULL;
17    }
18
19  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- **Este es el corazón del algoritmo. En cada paso del lazo tomamos el menor de todo el frente, lo guardamos en el vector de salida y reemplazamos el elemento con el siguiente del archivo correspondiente.**
- **Solo se revisan los archivos activos (`files[j]!=NULL`).**
- **Si en uno se acaban los elementos se cierra y se desactiva.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Ordenamiento externo con merge-sort (cont.)

- **Finalmente eliminamos todos los archivos auxiliares.**
- **(Todo el código está en *extsort.cpp*, *VERSION > aed-1.9.28*).**

```
1  fclose(out);
2  for (int j=0; j<nfiles; j++) {
3    sprintf(tmp_file,"tmp-file-%d.dat",j);
4    unlink(tmp_file);
5  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Comparación de métodos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

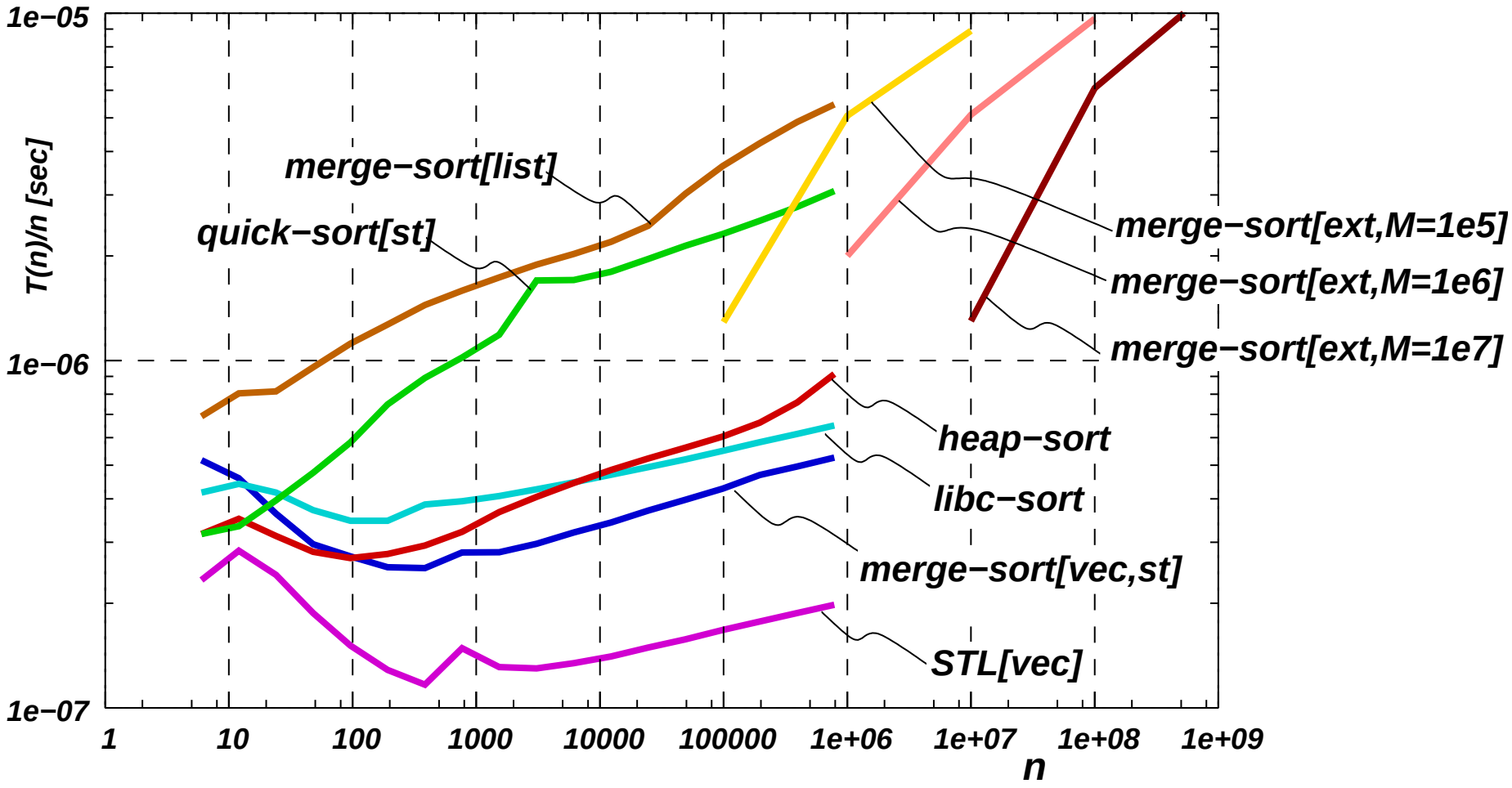
Comparación de métodos

Veremos una comparación de varias implementaciones de algoritmos de ordenamiento.

- ***STL[vec]*** **es la versión de sort que viene en el header** ***<algorithm>*** **de** ***C++*** **estándar con la versión de** ***g++*** **usada en este libro.**
- ***merge-sort[vec,st]*** **es la versión de merge-sort estable para vectores (no in-place).**
- ***libc-sort*** **es la rutina de ordenamiento que viene con el compilador** ***gcc*** **y que forma parte de la librería estándar de C (llamada** ***libc*****).**
- ***heap-sort*** **es el algoritmo de ordenamiento por montículos implementado en este libro.**
- ***quick-sort[st]*** **es el algoritmo de ordenamiento rápido, en su versión estable.**
- ***merge-sort[list]*** **es la implementación de** ***merge-sort*** **para listas.**
- ***merge-sort[ext,M=]*** **es la versión de merge-sort externa para varios valores del tamaño del bloque** ***M*****.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Comparación de métodos (cont.)



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Comparación de métodos (cont.)

Podemos hacer las siguientes observaciones

- **Para resaltar las diferencias entre los diferentes algoritmos se ha graficado en las ordenadas $T(n)/n$. Para un algoritmo estrictamente lineal (es decir $O(n)$) este valor debería ser constante. Como todos los algoritmos genéricos son, en el mejor de los casos, $O(n \log n)$ se observa un cierto crecimiento. De todas formas este cociente crece a los sumo un factor 10 o menos en 5 órdenes de magnitud de variación de n.**
- **Para algunos algoritmos se observa un decrecimiento para valores bajos de n (entre 10 y 100). Esto se debe a *ineficiencias* de las implementaciones para pequeños valores de n.**
- **El algoritmo de ordenamiento interno más eficiente de los comparados resulta ser la versión que viene con las STL. Esto puede deberse a que la implementación con template puede implementar *"inline"* las funciones de comparación. *NO es estable* (las STL tienen un `stable_sort()` que SI garantiza estabilidad).**
- ***Merge-sort para vectores resulta ser muy eficiente* (recordemos que además es estable, pero no es in-place). Sin embargo, para listas resulta**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

ser notablemente más lento. Esto se debe sobre todo a que en general la manipulación de listas es más lenta que el acceso a vectores.

- ***La versión estable de quick-sort es relativamente ineficiente*, sin embargo es el único algoritmo rápido, estable e *in-place* de los discutidos (recordemos que es $O(n(\log n)^2)$).**
- **Los algoritmos de ordenamiento interno se ha usado hasta $n = 10^6$. El *merge-sort* se ha usado hasta cerca de $n = 10^9$. Notar que a esa altura el tamaño de los datos ordenados es del orden de 4 GBytes.**
- **El algoritmo de *ordenamiento externo* parece tener una velocidad de crecimiento mayor que los algoritmos de ordenamiento interno. Sin embargo esto se debe a la influencia del tamaño del bloque usado. Para valores de n mucho mayores que M el costo tiende a desacelerarse y debería ser $O(n \log n)$.**

Diseño de algoritmos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Conceptos generales en diseño de algoritmos

No hay ninguna regla general para escribir programas y algoritmos. Todavía hoy el escribir programas sigue siendo en gran parte un *arte*. Sin embargo, con el transcurso del tiempo se han desarrollado una serie de técnicas que permiten ayudar a *diseñar algoritmos*, o al menos poder *clasificar* los diferentes algoritmos disponibles.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Dividir para vencer

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Algoritmos dividir-para-vencer

Probablemente esta sea la estrategia más simple de aplicar y entender. Básicamente se trata de *dividir un problema en problemas más pequeños*, hasta que, en el límite, los subproblemas son tan pequeños que su solución es trivial. Una condición esencial para que estos algoritmos sean aplicables es que la solución de los subproblemas sirvan de alguna manera a la solución del problema global.

Un ejemplo típico son los algoritmos *merge-sort* y *quick-sort*. En ambos el ordenamiento de una secuencia es realizada dividiendo primero a la secuencia en dos subsecuencias más cortas y aplicando el algoritmo *recursivamente* a cada una de ellas.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

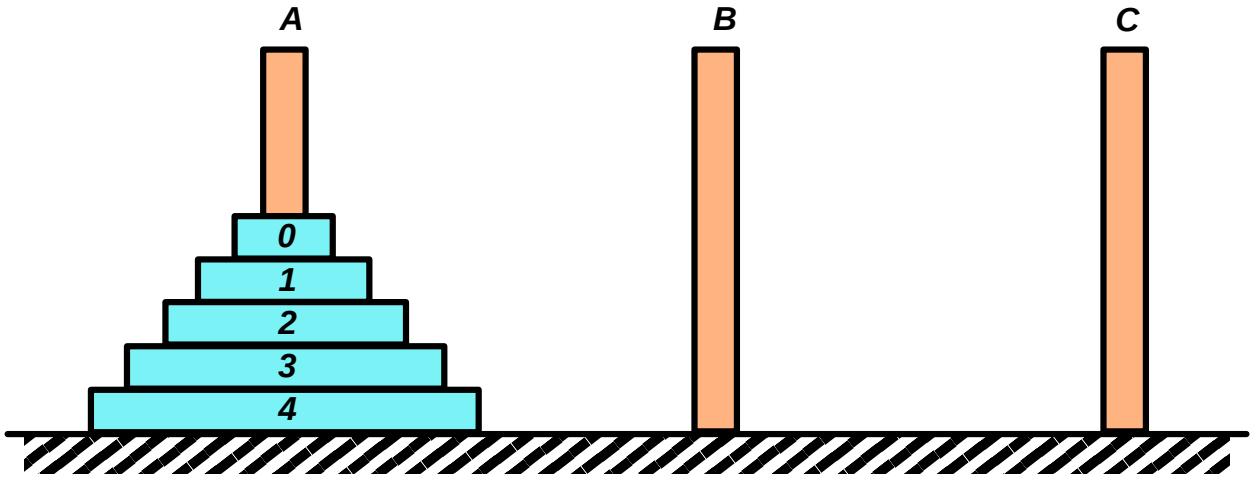
El problema de las Torres de Hanoi

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi

Consiste en tres *torres* (que llamaremos A, B y C) en forma de postes y una serie de n *discos* de tamaño creciente con un orificio en el centro, de tal manera que encajan en los postes. Denotaremos a los discos con un número de 0 a $n-1$. Inicialmente todos los discos están en uno de los postes (digamos el A) y el objetivo es *moverlos a otro de los postes* (B o C). (ver animación en YoutUbe)

- **Sólo se puede mover de un disco a la vez y**
- **en ningún momento puede haber un disco mayor arriba de un disco menor.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Por ejemplo tenemos las siguientes soluciones,

Para $n = 2$

`A:1 0`	`B:`	`C:`
`A:1`	`B:0`	`C:`
`A:`	`B:0`	`C:1`
`A:`	`B:`	`C:1 0`

Para $n = 3$

`A:2 1 0`	`B:`	`C:`
`A:2 1`	`B:0`	`C:`
`A:2`	`B:0`	`C:1`
`A:2`	`B:`	`C:1 0`
`A:`	`B:2`	`C:1 0`
`A:0`	`B:2`	`C:1`
`A:0`	`B:2 1`	`C:`
`A:`	`B:2 1 0`	`C:`

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Para $n = 4$

```
A:3 2 1 0     B:            C:
A:3 2 1       B:0           C:
A:3 2         B:0           C:1
A:3 2         B:            C:1 0
A:3           B:2           C:1 0
A:3 0         B:2           C:1
A:3 0         B:2 1         C:
A:3           B:2 1 0       C:
A:            B:2 1 0       C:3
A:            B:2 1         C:3 0
A:1           B:2           C:3 0
A:1 0         B:2           C:3
A:1 0         B:            C:3 2
A:1           B:0           C:3 2
A:            B:0           C:3 2 1
A:            B:            C:3 2 1 0
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El juego fue propuesto por el matemático francés *Edouard Lucas* en 1883. La leyenda dice que existe un templo en India donde los monjes mueven permanentemente un juego de *64 discos* y que *el mundo terminará* cuando se termine de completar el juego. (*De hecho veremos más abajo que esto es cierto!!*).

Si proponemos un algoritmo para resolver el problema, nos planteamos las siguientes preguntas

- **¿Da el algoritmo una *solución* al problema?**
- **¿*Como es el número de operaciones* (movidas) en función de n?**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Un algoritmo posible es el siguiente

- **a) *Mover el disco más chico* en sentido *cíclico* ($A \to B \to C \to A$)**
- **b) Hacer la *única movida posible* que no involucre al disco más chico.**
- **Volver a a) y repetir hasta que todos los discos queden en B o C.**

Por ejemplo, en las siguientes movidas

```
A:3 0          B:2 1          C:
A:3            B:2 1 0        C:
A:             B:2 1 0        C:3
```

primero movemos el disco más chico del poste A al B siguiendo la regla a). Después la regla b) indica que sólo podemos realizar una movida que involucre a los postes A y C. La única movida posible es pasar el disco 3 del poste A al C, ya que el poste C está vacío.

El problema de las Torres de Hanoi (cont.)

Para aplicar la *regla b),* notemos que

- ***Si los dos postes están vacíos,* entonces el algoritmo ha terminado, ya que deberían estar todos los discos en el restante.**
- ***Si uno de los postes está vacío,* entonces hay que pasar el tope del otro poste al vacío.**
- ***Si los dos postes tienen discos,* entonces hay que mover, de los dos discos en los topes, el menor, ya que mover el mayor involucraría violar la regla.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El algoritmo funciona, pero es difícil entender porqué. Tampoco no hay ningún indicio de en cuantas movidas se resuelve el problema.

Para $n = 3$

```
A:2 1 0     B:          C:
A:2 1       B:0         C:
A:2         B:0         C:1
A:2         B:          C:1 0
A:          B:2         C:1 0
A:0         B:2         C:1
A:0         B:2 1       C:
A:          B:2 1 0     C:
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El algoritmo es simple de programar. Tenemos a los postes como vectores de enteros en un vector de vectores `vector<vector<int> > vv`, de manera que `vv[0]` es el poste A, `vv[1]` es el poste B, etc... Los elementos están ordenados de mayor a menor, es decir que el más grandes está en el comienzo de la lista y el más chico en el fondo. El tope del poste es el fondo del vector, al cual se puede acceder con las funciones de la clase `vector<int>`: `x = back()`, `pop_back()`, `push_back(x)`, las cuales tienen un funcionamiento equivalente al del *TAD PILA*.

```
1  int n = 4;
2  vector< vector<int> > vv(3);
3  for (int j=n-1; j>=0; j--)
4    vv[0].push_back(j);
5  hprint(n,vv);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Una vez inicializados los postes, un lazo infinito comienza a hacer alternativamente las operaciones a) y b) descriptas arriba.

```
1 while (1) {
2   // Avanza menor ciclicamente . . .
3   if(vv[minor].size()==n) break;
4   hprint(n,vv); moves++;
5
6   // Realiza unica operacion que no
7   // cambia al menor . . .
8   hprint(n,vv); moves++;
9 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El código para avanzar el menor se basa en mantener una variable `int minor` que indica cuál de los postes contiene el menor. Inicialmente `minor=0`.

```
1    // Avanza menor
2    int next = (minor+1) % 3;
3    vv[minor].pop_back();
4    vv[next].push_back(0);
5    minor = next;
6    hprint(n,vv); moves ++;
7    if(vv[minor].size()==n) break;
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El código para la operación b) (la única operación que no afecta al menor disco) se implementa así

```
1     // Unica operacion que no cambia al menor
2     int lower = (minor+1)%3;
3     int higher = (minor+2)%3;
4     if (vv[higher].empty()) { }
5     else if (vv[lower].empty() ||
6           vv[lower].back() > vv[higher].back()) {
7       int tmp = lower;
8       lower = higher;
9       higher = tmp;
10    }
11    // Pasar el tope de 'lower' a 'higher'
12    int x = vv[lower].back();
13    vv[lower].pop_back();
14    vv[higher].push_back(x);
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

El código completo es como sigue

```
1  #include <cstdio>
2  #include <vector>
3
4  using namespace std;
5
6  void hprint(int n,vector< vector<int> > &vv) {
7    for (int j=0; j<3; j++) {
8      printf("%s:",(j==0? "A" : j==1? "B" : "C"));
9      vector<int> &v = vv[j];
10     vector<int>::iterator q = v.begin();
11     while (q!=v.end()) printf("%d ",*q++);
12     for (int j=v.size(); j<n; j++) printf(" ");
13     printf(" ");
14   }
15   printf("\n");
16 }
17
18 int main() {
19   int n = 4;
20   vector< vector<int> > vv(3);
21   for (int j=n-1; j>=0; j--)
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

```
22      vv[0].push_back(j);
23    hprint(n,vv);
24
25    int minor = 0;
26    int moves = 0;
27    while (1) {
28      // Avanza menor
29      int next = (minor+1) % 3;
30      vv[minor].pop_back();
31      vv[next].push_back(0);
32      minor = next;
33      hprint(n,vv); moves ++;
34      if(vv[minor].size()==n) break;
35
36      // Unica operacion que no cambia al menor
37      int lower = (minor+1)%3;
38      int higher = (minor+2)%3;
39      if (vv[higher].empty()) { }
40      else if (vv[lower].empty() ||
41               vv[lower].back() > vv[higher].back()) {
42        int tmp = lower;
43        lower = higher;
44        higher = tmp;
45      }
46      // Pasar el tope de 'lower' a 'higher'
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

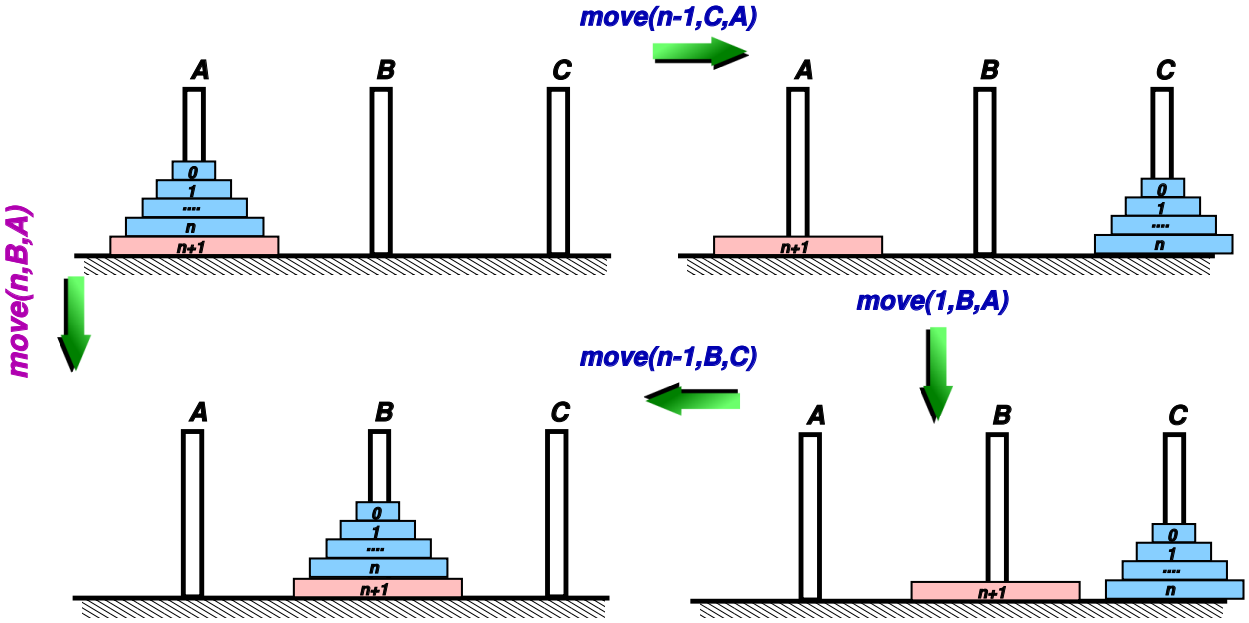
```
47      int x = vv[lower].back();
48      vv[lower].pop_back();
49      vv[higher].push_back(x);
50      hprint(n,vv); moves ++;
51
52    }
53    printf("solved puzzle in %d moves\n",moves);
54  }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Ahora consideremos la siguiente solución recursiva, basada en una estrategia *"dividir-para-vencer"* (*divide-and-conquer*). Si sabemos pasar n discos de un poste a otro entonces podemos pasar $n+1$ de A a B la siguiente forma

- **Mover los n discos menores (0 a $n-1$) de A a C.**
- **Mover el disco más grande (n) de A a B.**
- **Mover los n discos menores (0 a $n-1$) de C a B.**



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

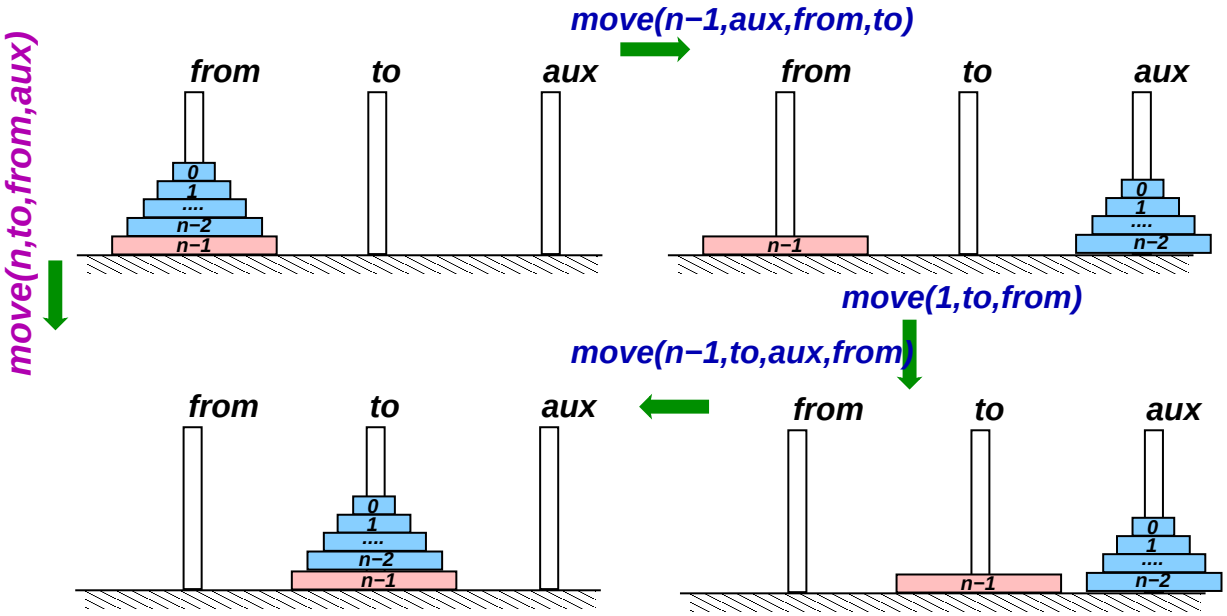
El problema de las Torres de Hanoi (cont.)

De esta forma *dividimos* el problema de mover $n+1$ discos en dos problemas más simples: mover n discos y mover 1 disco. El algoritmo es posible gracias a que, al mover los k discos menores de un poste a cualquier otro, los postes mayores ($k...n$) no obstaculizan el movimiento de los menores.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

```
1 void move(int n,vector<int> &to,
2           vector<int> &from, vector<int> &aux) {
3   if (n==1) move1(to,from);
4   else {
5     move(n-1,aux,from,to);
6     move1(to,from);
7     move(n-1,to,aux,from);
8   }
9 }
```



El problema de las Torres de Hanoi (cont.)

```
1 void move1(vector<int> &to,vector<int> &from) {
2   int x = from.back();
3   from.pop_back();
4   to.push_back(x);
5   hprint(vv);
6   moves++;
7 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

```
1  int main() {
2    int n = 10;
3    vv.resize(3);
4    for (int j=0; j<n; j++)
5      vv[0].push_front(j);
6    hprint(vv);
7
8    move(n,vv[1],vv[0],vv[2]);
9    printf("Solved Hanoi puzzle in %d moves\n",moves);
10 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

El problema de las Torres de Hanoi (cont.)

Se puede fácilmente contar el número de movidas

$$M(n) = 1 + 2M(n-1)$$

De manera que

$$M(2) = 1 + 2M(1) = 3$$

$$M(3) = 1 + 2M(2) = 7$$

$$M(4) = 1 + 2M(3) = 15$$

$$\dots$$

$$M(n) = 2^n - 1$$

Para el templo de la leyenda ($n = 64$, por lo tanto $2^{64} - 1 = 1.84 \times 10^{19}$ movidas) el tiempo necesario para resolver el juego es de (a 1 segundo por movida) *585 mil millones de años* (unas 42 veces la edad estimada del universo). Si lo hiciéramos en forma virtual en una PC actual (ciclo de la máquina $O(1\text{ns})$) entonces *apenas tardaría 585 años*.

El problema de las Torres de Hanoi (cont.)

La solución recursiva

- **Es fácil de entender porqué funciona.**
- **Permite obtener el número de movidas necesario.**
- **Aplica un criterio general, aplicable a una gran variedad de problemas (*dividir-para-vencer*).**

De hecho, las dos soluciones son equivalentes, sólo *difieren en la forma del planteo*.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Fixture de torneos todos-contra-todos

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Fixture de torneos todos-contra-todos

Otra aplicación de la estrategia *"dividir-para-vencer"* es el algoritmo para diseñar torneos *"todos-contra-todos"*. El mismo algoritmo se puede aplicar para programar el intercambio de datos en operaciones *"all-to-all"* en cálculo distribuido.

- **Tenemos una serie de n jugadores $P_0, ..., P_{n-1}$ y tenemos que hacerlos jugar a todos contra todos, en una serie de fechas.**
- **En cada fecha un jugador puede jugar un solo partido y debemos tratar de armar el *"fixture"* de manera de tener el menor número de fechas posibles.**
- **Tenemos $n(n-1)/2$ partidos y al menos debe haber $n-1$ fechas ya que cada jugador debe jugar $n-1$ partidos y no puede jugar más de un partido en una fecha.**

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

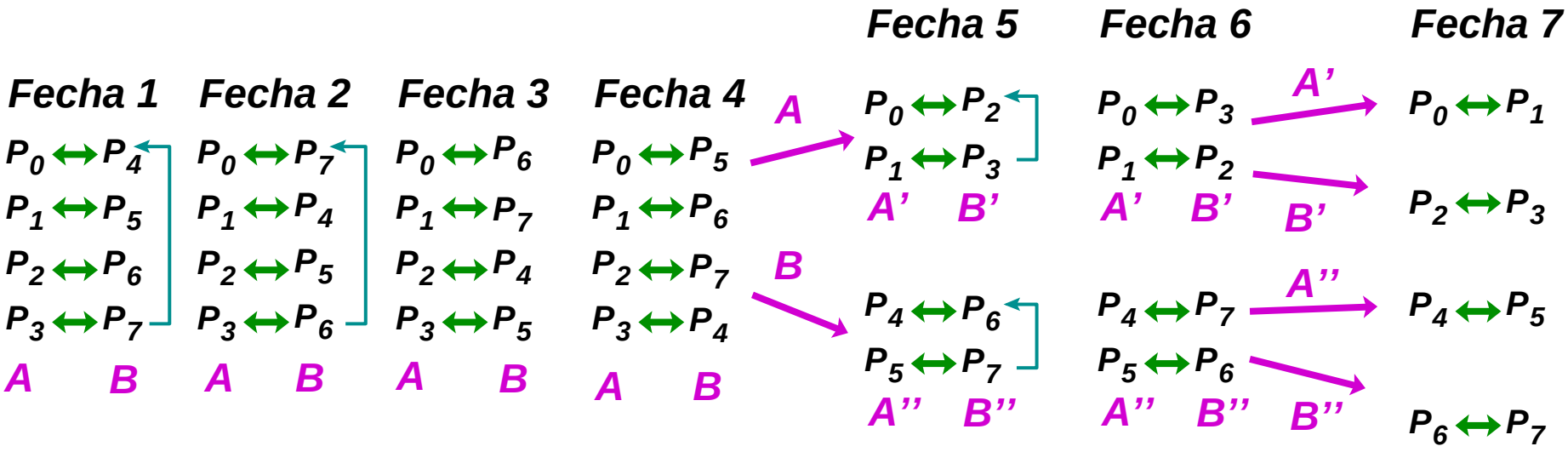
Fixture de torneos todos-contra-todos (cont.)

Pensando en la estrategia *dividir-para-vencer*, si dividimos al conjunto de n jugadores en *dos partes iguales* (los llamaremos A y B) y pudiéramos resolver los partidos internos dentro de cada grupo *en forma recursiva*, entonces sólo falta programar los partidos entre jugadores del subgrupo A y del subgrupo B. Pero esto se puede hacer sencillamente haciendo jugar los jugadores de A con los de B en un cierto orden en la primera fecha, y después en las siguientes $n/2 - 1$ fechas *rotando cíclicamente* los jugadores de B.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Fixture de torneos todos-contra-todos (cont.)

El algoritmo es más simple de visualizar para $n = 2^m$. Sea por ejemplo $n = 8$:



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Fixture de torneos todos-contra-todos (cont.)

El número de fechas se puede calcular en forma *recursiva*

$$F(n) = n/2 + F(n/2)$$

que aplicando recursivamente da

$$F(2) = 1$$

$$F(4) = 2 + F(2) = 3$$

$$F(8) = 4 + F(4) = 7$$

$$\vdots = \vdots$$

$$F(n) = n - 1$$

que es la *solución óptima*, ya que vimos que al menos hacían falta $n-1$ fechas. Notar que este problema es equivalente a *colorear un grafo*, pero en ese caso o bien podemos aplicar un *algoritmo exhaustivo* (que es $O(n!)$ *no-polinomial*) o bien uno *exhaustivo* que no garantiza la solución óptima.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Divide-and-conquer no siempre puede aplicarse

Pensemos por ejemplo en el problema del agente viajero (TSP). Podríamos plantearlo en términos de divide-and-conquer dividiendo las ciudades en dos subgrupos y hallando la solución óptima para cada subgrupo. Pero entonces deberíamos tratar de obtener el camino óptimo para todas las ciudades *combinando los dos subcaminos óptimos hallados*. Ahora bien, el camino óptimo global puede no tener ninguna relación con los subcaminos óptimos. Por lo tanto aquí *"divide-and-conquer"* no aporta ninguna solución interesante, al menos desde esta óptica.

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Programación dinámica

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Técnicas de programación dinámica

En el proceso de aplicar la estrategia de *"dividir-para-vencer"* en forma repetida, la cantidad de subproblemas pequeños que se van creando *puede empezar a crecer en forma exponencial*, terminando en algoritmos con complejidad algorítmica no polinomial. Sin embargo es posible que en realidad este crecimiento sea artificial ya que algunos de los subproblemas que se van creando por las diferentes ramas sean en realidad el mismo. En este caso puede ser útil *almacenar una tabla con los resultados parciales* de los subproblemas.

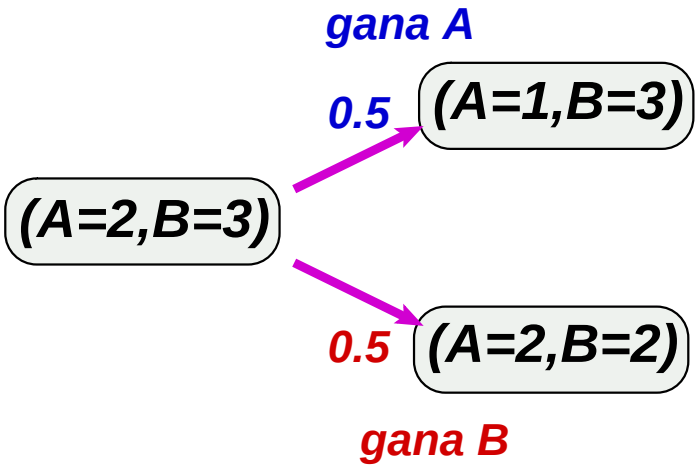
((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial

Tenemos dos equipos A y B que juegan una serie de partidos hasta que alguno de ellos gana n partidos. Por ejemplo, si $n = 4$ y A gana el primer, partido B el segundo y siguiendo A, B, A, A (denotamos la secuencia total como A, B, A, B, A, A), entonces termina ganando la serie mundial A ya que acumuló primero 4 partidos ganados. Supongamos que en cada partido la probabilidad de que gane A o B sea la misma. Entonces si en cierto momento A lleva ganados 3 partidos y B lleva ganados 0, es mucho más probable que A gane la serie mundial, ya que le falta un solo partido por ganar, mientras que a B le faltan 4.

Cálculo de la prob. de ganar la serie mundial (cont.)

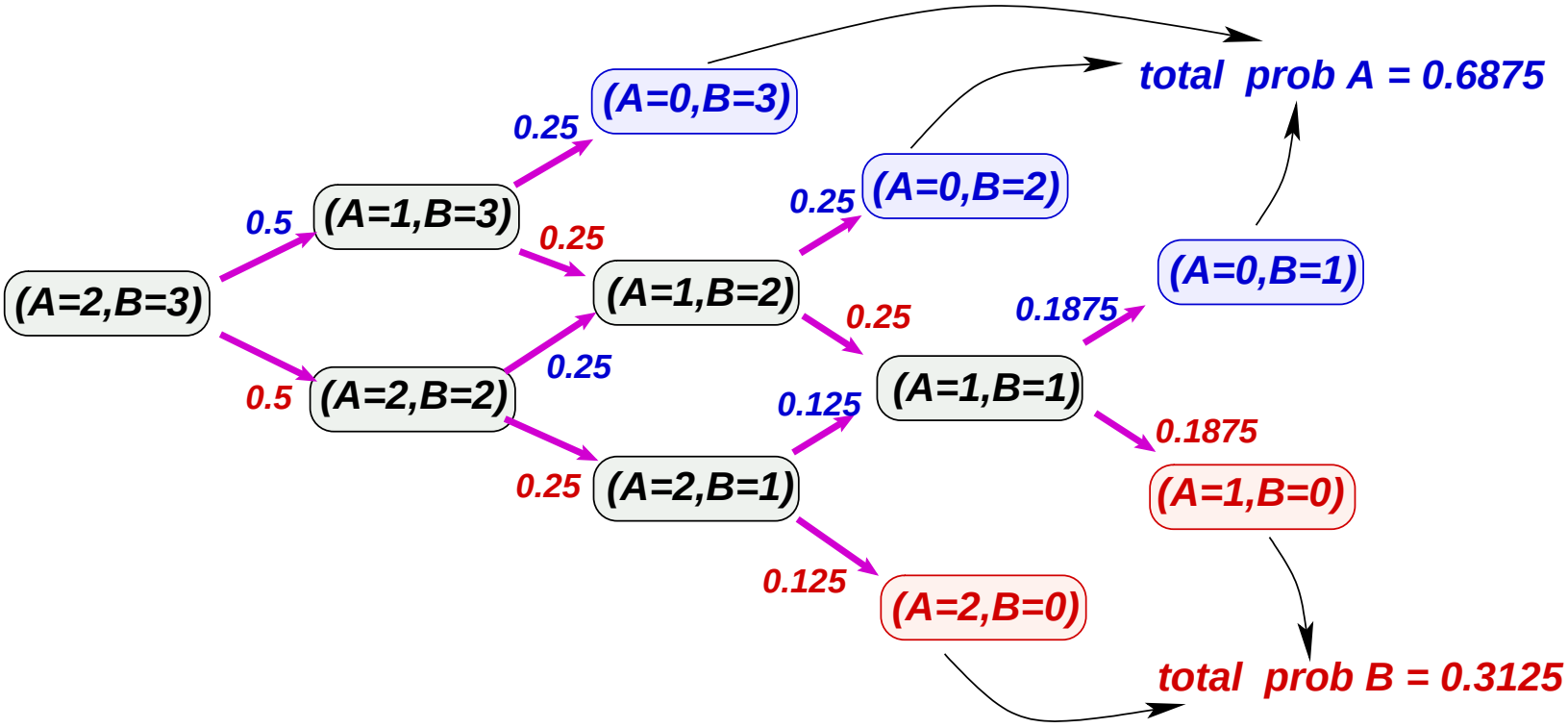
Denotamos los estados en que puede desarrollarse el torneo como $(A = p, B = q)$ donde p y q son la cantidad de partidos que le falta ganar a A y B para ganar la serie. Inicialmente $(A = 1, B = 4)$. De allí la probabilidad de que A o B ganen es la misma, en cuyo caso pasarían a los estados $(A = 0, B = 4)$ o $(A = 1, B = 3)$ respectivamente. Cuando uno de los dos es 0, el torneo termina.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

Así se puede construir toda la tabla, hasta que alguno de los dos equipos gana. De esta forma se puede calcular la probabilidad de que cada uno de los equipos gane lo cual puede ser de utilidad a la hora de apostar ( ).



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

Otra forma de ver las cosas es la siguiente. Consideremos la relación estadística

$$P(x) = P(y)P(x|y) + P(z)P(x|z)$$

donde $P(x)$ es la probabilidad de que ocurra el evento x, el cuál se depende de dos eventos complementarios y y z. $P(x|y)$ es la probabilidad de que ocurra x si se produce el evento y y $P(x|z)$ la probabilidad de que ocurra x si se produce el evento z. Por ejemplo, la probabilidad de que salgan dos caras al lanzar dos monedas simultáneamente al aire es

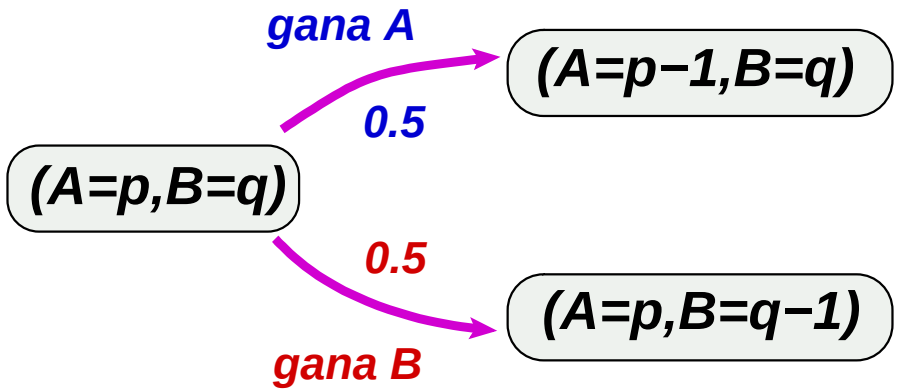
$$\begin{aligned} P(\textbf{2 caras}) &= P(\textbf{M1=cara})\, P(\textbf{2 caras}|\textbf{M1=cara}) \\ &\quad + P(\textbf{M1=ceca})\, P(\textbf{2 caras}|\textbf{M1=ceca}) \\ &= 0.5\, P(\textbf{2=caras}|\textbf{M1=cara}) \\ &= 0.5\, P(\textbf{M2=cara}) = 0.5 \cdot 0.5 = 0.25 \end{aligned}$$

((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

Entonces, si consideramos la probabilidad $P_A(p,q)$ de que gane A asumiendo de que a A le faltan p partidos y a B le faltan q, entonces tenemos la relación recursiva siguiente

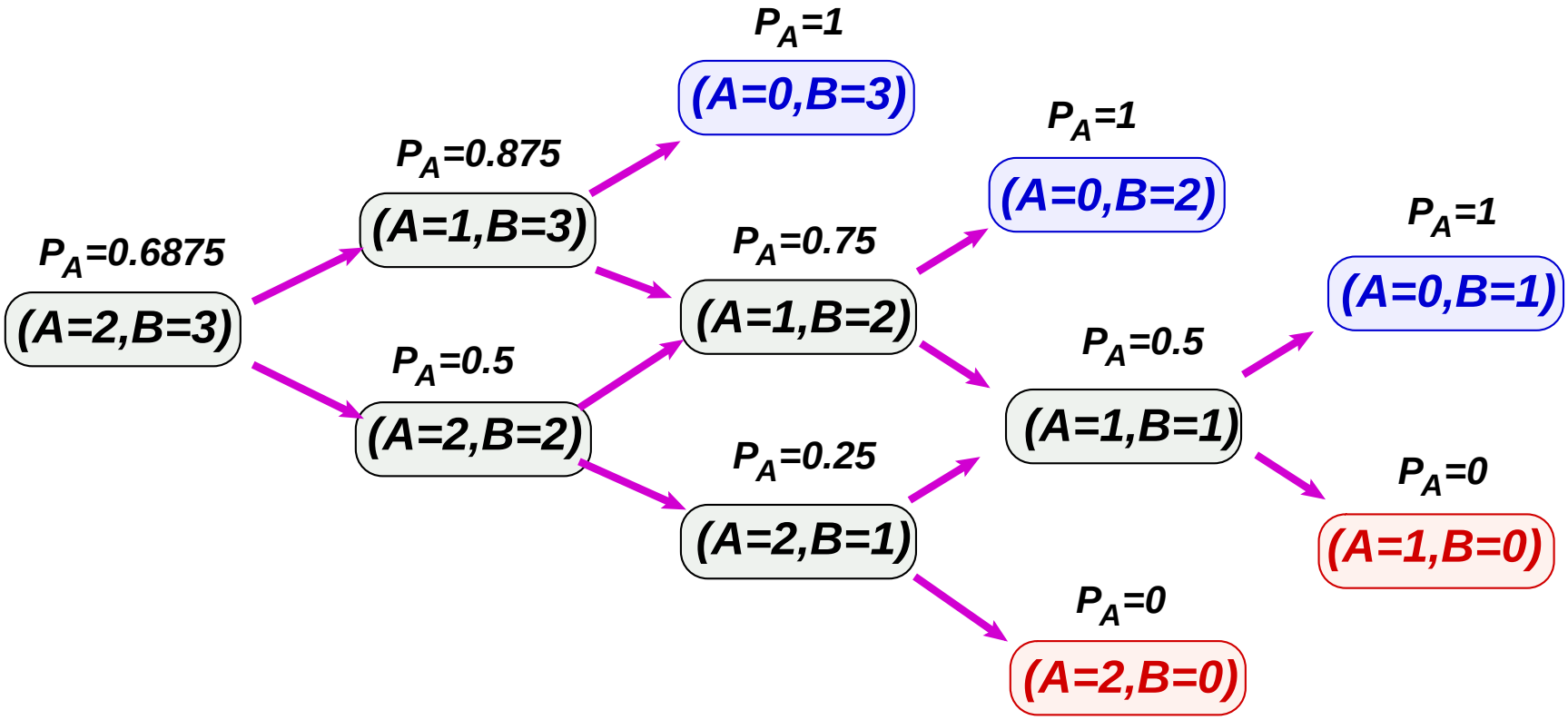
$$P_A(p,q) = P(\text{gane } A)\, P_A(p-1,q) + P(\text{gane } B)\, P_A(p,q-1)$$



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

Aplicando esta relación recursivamente construimos la siguiente tabla. Para cortar la recursión $P(0,q) = 1, P(p,0) = 0$. Vemos que obtenemos el mismo resultado $P_A(2,3) = 0.6875$.



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

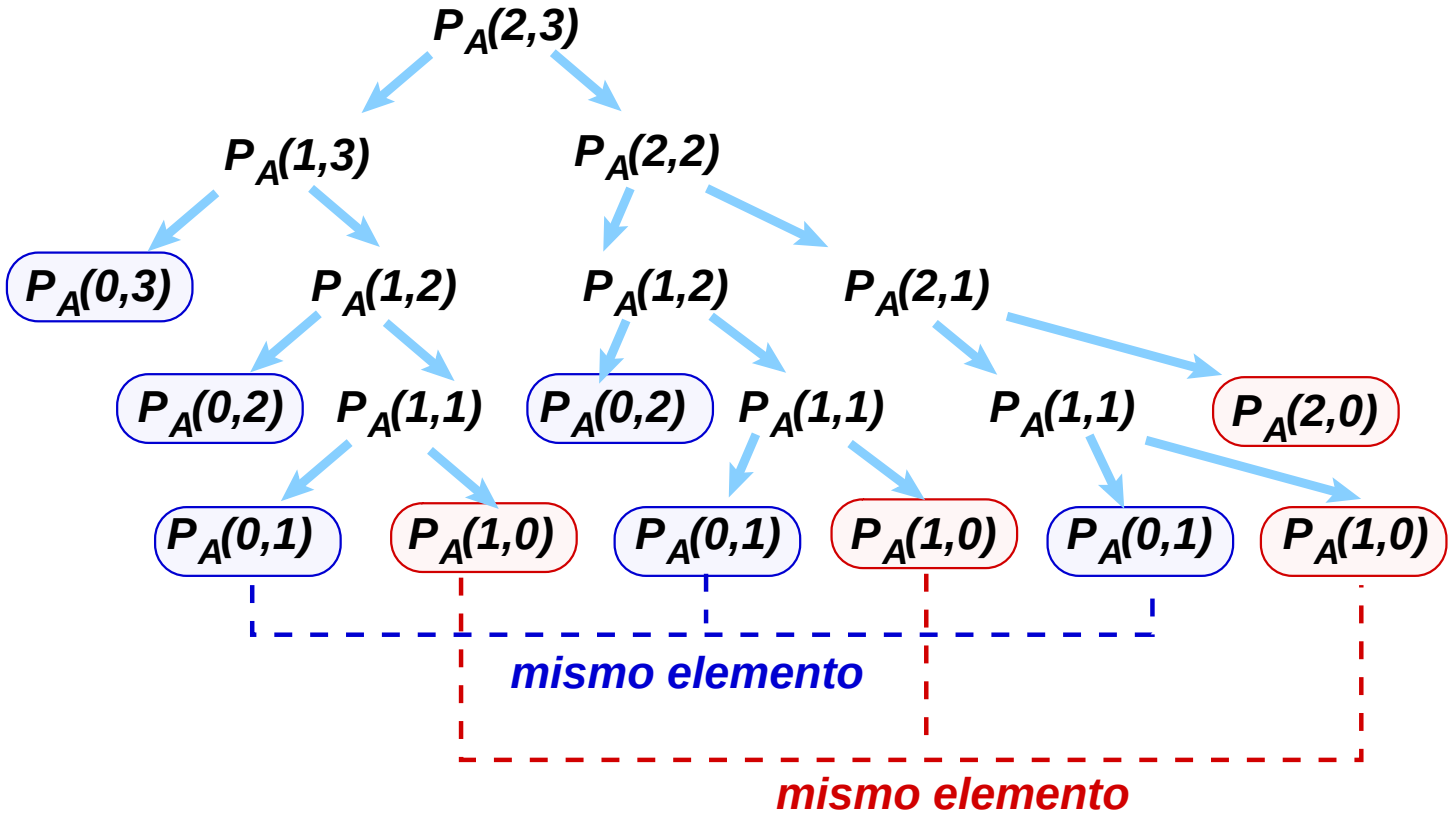
Es muy sencillo implementar una función recursiva que calcular $P_A(p,q)$:

```
1 double p_A(int p,int q) {
2   if (!p) return 1;
3   if (!q) return 0;
4   return 0.5*(p_A(p-1,q)+p_A(p,q-1));
5 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

Sin embargo, si dibujamos el árbol de llamadas recursivas que se produce, notamos que algunos elementos son calculados varias veces



((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))

Cálculo de la prob. de ganar la serie mundial (cont.)

A medida que crecen p y q el número de llamadas recursivas redundantes es cada vez mayor. Como el árbol binario que expande la llamada recursiva puede tener hasta $p + q$ niveles, el número de llamadas podría llegar a 2^{p+q}, cuando en realidad debe calcularse sólo para $p \cdot q$ pares.

Cálculo de la prob. de ganar la serie mundial (cont.)

La técnica de *"programación dinámica"* se basa en evitar la duplicación de llamadas almacenando en una tabla los valores de la función previamente calculado. También es un ejemplo de *"evaluación diferida"* (*"delayed evaluation"*). Por ejemplo, para $p = 8, q = 10$ la versión previa hace 43757 llamadas, mientras que esta implementación hace sólo 80. Algunos lenguajes funcionales de alto nivel proveen funciones específicas para estos casos (e.g. Scheme con *delay*, *force*, *promise? & friends*).

```
1  double p_A2(int p,int q) {
2    static map< pair<int,int>,double> table;
3    if (!p) return 1;
4    if (!q) return 0;
5    pair<int,int> Q(p,q);
6    map< pair<int,int>,double>::iterator w =
7      table.find(Q);
8    if (w!=table.end()) return w->second;
9    double prob = 0.5*(p_A2(p-1,q)+p_A2(p,q-1));
10   table[Q] = prob;
11   return prob;
12 }
```


((version aed-2.0.4-10-gdb5d209) (date <unknown>)
(processed-date Wed Dec 22 17:51:23 2010 -0300))